Computer Magic, Inc.

ATARI GAME OWNERS

You already own a powerful hobby computer.

Unleash the capabilities of the 6502 microprocessor inside your Atari Video Game Console with the MagiCard module, the first in a line of new generation plug-in modules and game cartridges from Computer Magic. The MagiCard is ideal either for those who've been looking for an inexpensive way to learn about computers and microprocessors, or for those who've been wanting to program their own TV games. In less than an hour you can be creating and running machine language programs that can make use of all the video and audio features of the Atari game.

The MagiCard is a printed circuit board of somewhat greater complexity than those found inside game cartridges. It plugs into the connector on top of your Atari Video Game Console just like an ordinary game cartridge. Unlike an ordinary game, however, the MagiCard allows you full access to the 6502 microprocessor and video display generator inside your Game Console.

A new technique (patent applied for) allows the MagiCard to contain both ROM (read-only memory) and RAM (random-access memory). The 2K bytes of ROM contain a monitor program with such features as fetch from memory, store to memory, display memory contents on TV, display disassembled 6502 machine code on TV, run a program, and cassette read and write. 1K bytes of RAM allow for storage of user programs and data. Your keyboard controllers provide communication with the monitor and allow for entry of machine language programs. Monitor subroutines are also made available to the user, allowing easy display of alphanumeric text and memory-mapped graphics.

Included with the MagiCard is a 136-page instruction manual which can serve as an introduction to machine language programming for novices, and which contains details on the 6502 microprocessor, the MagiCard features, and the audio and video capabilities of the Atari game. Also provided are plans for building an inexpensive cassette interface. As a special bonus, the manual includes a sample program of the Game of Life, the popular computer hobbyist game first featured in Scientific American in Oct. 1970. This version of Life features a 40x42 grid and more than 200 generations per minute.

MagiCard plus manual are available for $49.88 (Illinois residents add 6% tax, please). The manual can be purchased separately for $15.00 (deductable from the subsequent purchase of one Magicard within 30 days). Order from:

Computer Magic, Inc.
PO Box 3383
Fox Valley Center
Aurora, IL 60505

MagiCard

Instruction Manual

MagiCard Instruction Manual

Computer Magic, Inc.
Box 3383
Fox Valley Center
Aurora, Illinois
60505

312-357-8064

Copyright 1981 by Computer Magic Inc.

Contents

A Note to the Reader

Introduction

Chapter 1 Computer Fundamentals

Chapter 2 Using Your MagiCard

Chapter 3 Description of 6502 Microprocessor

Chapter 4 MagiCard Keyboard Functions

Chapter 5 MagiCard Memory Map

Chapter 6 Details of Video Computer System Features

Chapter 7 MagiCard Monitor Subroutines

Appendix A Hexadecimal-Decimal Conversion Table

Appendix B MagiCard Character Set

Appendix C 6502 Instruction Summary

Appendix D "Life"--a Sample Program

Appendix E Construction and Connecting a Cassette Interface

Owner Information

A Note To The Reader

Congratulations on purchasing our MagiCard 6502 computer adapter card for your video game! First some words of caution: Never insert or remove the MagiCard or any controller connections to your game while it is turned on. Always handle the MagiCard by the edges and touch the game console switches with your free hand just before inserting it. Avoid touching the TV screen with your hands or any of the game cables to further minimize the chance of damage due to static electricity. Finally, be sure you know which side of the MagiCard is "up" (see Chapter 2), and always insert it right side up. Most of these precautions are helpful with conventional cartridges as well, and if you make a habit of them, your game should function for many years.

Depending on your experience with computers, and the 6502 in particular, you may be able to skip or skim the earlier chapters of this manual. Computer stores, bookstores and libraries often have a few of the many books written about the 6502. Don't try to read them all. Programming The 6502, third edition, by Rodnay Zaks is one of several good ones. Some aspects of the game system are either too detailed to describe properly in a manual of limited size, or so obscure that we aren't sure what to say. These unclear areas are never vital to a basic understanding of the game and are generally deducible from clues we give and from tests using the MagiCard itself. Happy exploring!

In one respect, programming the MagiCard - Video Computer combination may be a new (and educational) experience even for some experienced programmers. It is a "real-time" system, with the 6502 program handling all aspects of the display, sound and controller response on an instant by instant basis. The display, for example,

may be created line by line as the electron beam sweeps across the TV screen, requiring the program to reweave this evanescent tapestry every 1/60 second. Don't worry, the MagiCard makes it very easy to create certain displays, so you can start programming quickly. And it is possible, with effort, to program displays and games rivaling those available in cartridges. Welcome to the real (time) world.

Computer Magic 1981

Introduction

Inside your Atari Video Computer System is a powerful 6502 microprocessor, a "computer on a chip." With this manual and your MagiCard, you will learn to program this tiny computer and explore the full capabilities of your Atari game.

Chapter 1 will introduce you to some fundamental concepts about computers, including the ideas of a program, memory, and the hexadecimal number system--the language you will use to speak to your computer. If you are already familiar with these concepts, you may start with Chapter 2.

Chapter 2 will show you how to get started with your MagiCard. By working through a few simple examples, you will learn how to write programs for your computer, how to enter programs and data into memory, and how to run your programs. You will learn to produce displays on your television screen and sounds through your television speaker.

The last five chapters in the manual provide a variety of detailed information about the MagiCard and the Atari Video Computer System. Chapter 3 gives a detailed description of the 6502 microprocessor. Chapter 4 covers the use made by the MagiCard of each key on your keyboard controllers. The memory map for the MagiCard is the subject of Chapter 5. Chapter 6 describes the detailed workings of the Atari game features. Finally, Chapter 7 describes the MagiCard monitor subroutines that are of potential use to you in your programs.

CHAPTER 1

Computer Fundamentals

All computers, from the smallest single-chip microprocessor to large machines that fill up a room, have certain features in common. Computers, although fast, accurate, and powerful, are really rather dumb. The only thing they can do is to execute a series of extremely simple instructions. Examples of such instructions are to add two numbers together, to move a number from one place to another, and to check if one number is larger than another.

Also, all computers have three basic parts. These are

1. The central processor unit (or CPU). This is the brain of the computer, the place where it interprets and executes the instructions. In your computer, the CPU is the 6502 microprocessor inside of your Video Computer System.

2. The memory. This is the place where the computer stores information. This information can be either a program, which is a list of instructions for the computer to perform, or data, which is a list of numbers for the program to operate on. In your computer, there is a small amount of memory inside of your Video Computer System and substantial additional memory on your MagiCard.

3. Input-output devices. These are the computer's eyes, ears,

and mouth, the way it receives information (both programs and data) from the outside world, and the way it transmits its answers to you. In your computer, your keyboard controllers are the input device, and your television screen and television speaker the output devices. If you build the audio cassette interface, a cassette recorder can be used as both an output and input device for programs and data.

Any problem you wish to solve on a computer, whether it be balancing your checkbook, playing a game, or forecasting the weather involves the same series of steps:

1. You must write a program, a list of instructions telling the computer what to do.

2. You must enter the program into the computer memory by using an input device.

3. You must enter into the computer memory the data that the program will operate on. (For example, consider balancing a checkbook. The program would be a list of instructions telling the computer to add deposits and subtract checks from your balance. The data would be a list of that month's opening balance, deposits, and checks.)

4. You run the program. The computer executes your program, doing each instruction in order, and then displays the answers on its output device (provided you have included an instruction in your program telling it to do so).

Your MagiCard will enable you to follow these steps to perform a wide variety of tasks on your Video Computer System. Before going on to specific details of the use of your MagiCard, we must first discuss a few more topics about computers in general. In particular, you will need to understand the hexadecimal number system, which is the language you will use to talk to your computer. You will need to understand a little more about the structure of a computer so that you will know what kinds of instructions you can give it.

First consider the hexadecimal (or hex) number system. In this manual numbers are in hex unless written out or expressly stated to be in decimal. Hexadecimal is a number system based on 16, rather than our usual decimal number system based on 10. In our normal system, for example, the number 23 represents 2 tens plus 3 ones, or the quantity twenty-three. In hex, on the other hand, the digits 23 represent 2 sixteens and 3 ones, or the quantity thirty-five (see Figure 1).

Figure 1

Furthermore, a new way is needed to represent the quantities ten through fifteen with a single digit, and we choose to use the letters A, B, C, D, E, and F for this purpose. Thus, to count from one to

twenty in decimal and hex would look like:

decimal:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
hex:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14

Hex is the language that your computer uses. You will need to be able to recognize that AB is a number, and that the number that comes after AB is AC. To find the decimal value of AB you can either look it up in the chart in Appendix A, or calculate it yourself as follows:

AB = "A" sixteens + "B" ones
= 10 sixteens + 11 ones
= 160 + 11
= 171

Now let's look in more detail at what is inside a computer. First, consider the memory. You should think of the memory as an ordered series of locations like the houses on a street. In each location you can store either instructions for the computer or data. Moreover, again like the houses on a street, each location in memory has an address. The address of the first location in memory is zero, and each subsequent location has an address one higher than the previous location. These addresses are hex numbers. For example, as in Figure 2, the first several locations in memory have addresses 00, 01, 02, 03, etc., while later on in memory there is a group of locations with addresses A9, AA, AB, AC, etc.

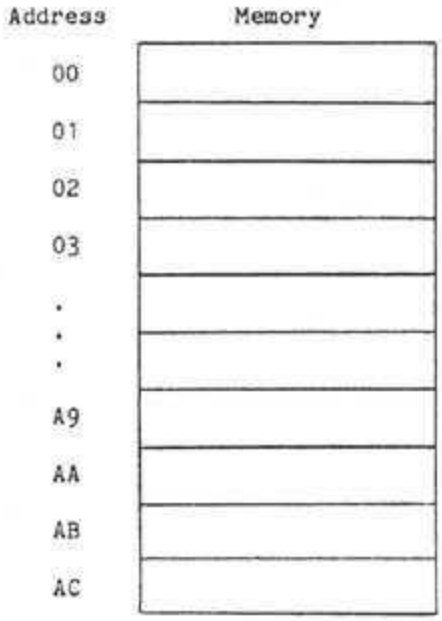


Figure 2

The address of a location of memory is very important because it is the way that the computer finds the correct instruction to execute or the right piece of data to use (just like you use the address of a house to find the right house on a strange street).

Now you can understand the order in which the computer executes its instructions. It goes through the memory and executes the instructions in order (like a salesman knocking on the door of one house after another on the street). For example, you might tell the computer to start by executing the instruction in memory location A7. It will then continue with the instruction at A8, then A9, AA, etc. The computer will proceed along in this manner until it finds an instruction telling it to stop, or a special kind of instruction (called a "jump") telling it to jump off and start executing instructions in sequence starting at a new location. For example, consider the part of a program shown in Figure 3.

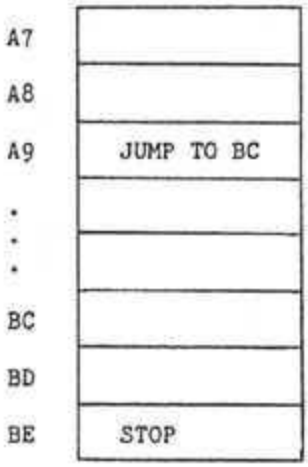


Figure 3

If you start the computer at address A7, it will execute in order the instructions at A7, A8, A9 (which causes it to jump), BC, BD, and BE, where it will stop. In fact, one instruction can occupy several locations in sequence. That does not complicate the simple execution of locations in order.

Finally, we discuss just what the various instructions tell the computer to do. (You have already learned about two instructions in the above example: STOP and JUMP). To do this you must know a little bit about the central processor unit (CPU) of your computer. Inside the CPU are some special locations called "registers". Data is moved from locations in memory into these registers, where the data can be manipulated. A "LOAD" instruction moves data from memory to a register. A "STORE" instruction moves data from a register to memory. Other instructions operate on the data in a register; for example, "ADD" instructions add a number to the register and "SUB" instructions subtract a number from the register. You can understand this more clearly by following the state of memory and of a register after the execution of each instruction of a simple program. The program (see Figure 4) is designed to add the number stored in memory location C2

to a running total stored in loaction C1.

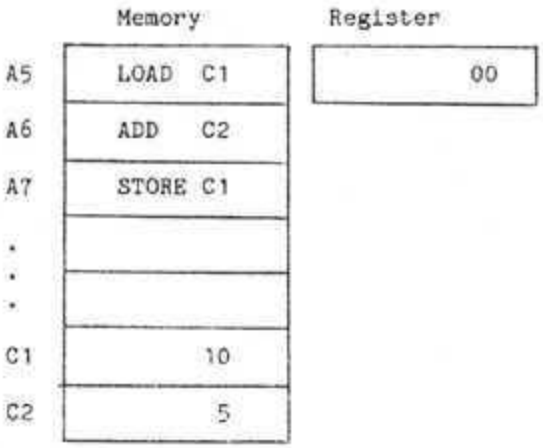


Figure 4

The initial contents of the register is zero and the first instruction to be executed by the computer is contained in memory location A5 (LOAD C1). After executing the first instruction (Figure 5a), the contents of location C1, namely the number 10, has been moved into the register.

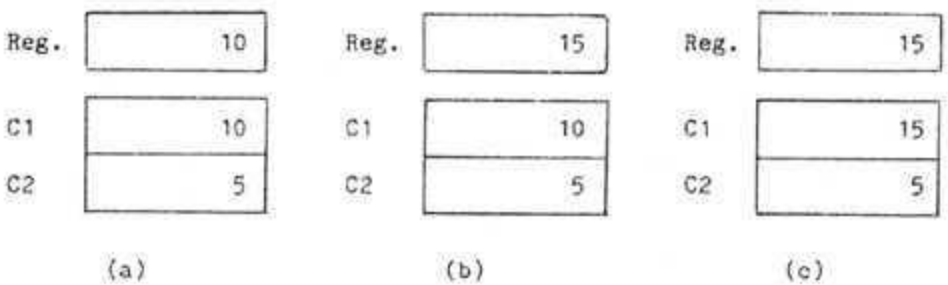


Figure 5

The computer then executes the instruction at location A6 (ADD C2), which causes the contents of location C2, the number 5, to be added into the register. Now the state of the computer is as shown in Figure 5b. Finally, the computer executes the instruction at location

A7 (STORE C1), which copies the contents of the register back into location C1. The new state of the computer is as shown in Figure 5c, with the sum stored back into location C1. The computer will continue executing instructions with whatever instruction is in location A8.

The general features just illustrated are common to all computers. Each model of computer will have at most a slightly different internal arrangement, with different kinds of registers and a different set of instructions. After you have understood the material in this chapter, you are ready to go on to learn about the specific instructions available in your computer, the 6502 microprocessor, and to learn how to use your MagiCard.

CHAPTER 2

USING YOUR MAGICARD

2.1 PLUGGING IN AND TURNING ON

The first step is to assemble your MagiCard system, which is little more difficult than plugging any game cartridge into the game console. With the On/Off switch off, insert your MagiCard into the large slot centered in the cartridge part of the game console. You will need to stick the small sliver of fiberglass board provided into one of the small slots on either side of the larger slot to open the door behind the large slot. Be sure the component side of the MagiCard--the side with the plastic electronic parts--is facing up. It is a good idea to always handle your MagiCard by the side edges. Make sure it is firmly seated in the slot. Next, plug in the two keyboard controllers. (If you do not already own a set, keyboard controllers are available at most stores selling Atari cartridges.) Slide the two controllers together, making sure the controller on the left-hand side is plugged into the left-hand part of the game console. For your convenience, two cardboard templates giving the functions of the keys have been provided. If you wish, you can place these over the keys of the controllers.

Now, turn your game on. The letters cMx-2 should appear in the middle of your TV screen, and there should be a small square blinking about once per second near the top. cMx-2 identifies the MagiCard (if you buy other products from us they will identify themselves

differently). The blinking square indicates the MagiCard is ready to accept your commands. (If you do not see this display, make sure your game console has its power cord connected and plugged in, that it is attached to your TV set, that the switch near the TV is in the game position, that your TV is tuned to the proper channel, and that the MagiCard and keyboard controllers are firmly plugged in.)

Now you are ready to start learning the function of the various keys on the keyboard controllers. First, consider the 16 keys shown in figure 6,

1	2	3
4	5	6
7	8	9
	0	

A	B	C
D	E	F

Figure 6

used for the 16 hex digits. Try pressing some of these keys one at a time. (Don't press any other keys yet.) As you press them, you will hear a beep and see the hex digits appear on the upper right of your TV screen (ignore the number F0F0 which appears on the left--it has no meaning yet). Practice entering various hex numbers. This is the way you will enter instructions and data into your computer. If you make a mistake, just start over again pressing the correct numbers. They will replace the incorrect ones, as you can observe on your TV.

2.2 STORING NUMBERS IN MEMORY

Now that you know how to get hex numbers up on the TV screen, you are ready to learn how to store numbers in the computer's memory. You will first have to tell the computer the address of the memory location you wish to store a number into, and then tell it the number you wish to store. To do this you use the two new keys shown in Figure 7,

X	X	X
X	X	X
X	X	X
	X	STORE

X	X	X
X	X	X
X	X	X
	X	ST. AD.

Figure 7

the "Store Address" (ST. AD.) and "Store" keys. (The X's in figure 7 represent keys you have already learned about.)

The sequence of steps to load hex numbers (which can represent either instructions or data) into the computer's memory is as follows:

1. Type in the 4-digit hex number of the address you want to store into. It appears at the upper right of the TV screen. (For example, type in F030.)

2. Press the "Store Address" key. The only visible effect this will have will be to change the number at the upper left of the TV screen from F0F0 to 0000. More importantly, this tells the computer that the number you just typed in, in this case F030, is the address of a location in memory into which you intend to store a number.

3. Type in the 2-digit hex number you wish to store. For example, type 55. The upper right of the TV should now read 3055 (The 30 is left over from the F030 you typed in Step 1. Only the 55 is important, because a single memory location is only large enough to contain two hex digits. Only the last two digits count in these circumstances, and so the 55 will be stored and the 30 ignored).

4. Press the "Store" key. This is what actually causes the number 55 to be stored into memory location F030. This should cause two effects on your TV screen: first, F030 should appear at the upper left as a reminder that you have

just stored into that location; and second, a pattern of 4 dots should appear just under F030. The four dots appear because the data in memory location F030 is used to generate the picture at one position on the TV screen. Originally this location contained 00 and so that area of the TV screen was blank; but when you stored 55 into that location you caused a pattern to appear on the TV. (In the last section of this chapter a more detailed discussion is given on how to draw whatever patterns you want on the TV screen.)

For more practice storing into memory repeat steps (1)-(4) using different memory locations (try F05A and F084 instead of F030) and different data (try 33 and FF instead of 55). You should see different patterns appear at different places on your TV. In each case, however, the address that you've stored into should appear at the upper left of the TV.

Certain memory locations in your computer have special functions. For example, memory location 0008 sets the color of objects appearing on your TV. As an exercise, try storing 85 into location 0008. (This involves going through Steps (1)-(4), typing in 0008 at Step 1 and 85 at Step 3.) When you press the "Store" key, the numbers on your TV will change fron pink to blue! (A list of all these special locations and their functions is given in later chapters of this manual.)

Now you are ready for some exercises:

Exercise 1: Turn the numbers on the TV back to pink. (The pink color corresponds to the number 55.)

Answer: Store 55 into location 0008.

You might like to experiment by storing different numbers into location 0008 and seeing the different colors your computer can produce. Be careful, however. If you store 00 into location 0008 the numbers will seem to disappear, because they will be drawn in the same color as the background! To make them reappear you will have to store a non-zero number into 0008. By the way, the color of the background

is determined by the number stored into memory location 0009; therefore, by storing into both 0008 and 0009 you can produce arbitrary combinations of colors.

Exercise 2: Erase the pattern of dots you created when you first stored numbers into memory.

Answer: Store 00 into F030, F05A, F084, and any other locations you stored a non-zero value into to clear those areas of the TV screen; or

Alternate answer: Press the "Game Reset" switch on your game console. This will clear off the screen, restore the original pink color, and give you a fresh start.

Finally, there is one more thing you need to know about storing numbers into memory which is especially important when you want to store into several memory locations in a row (as you will certainly do whenever you enter programs into the computer). Each time you press the "Store" key, the computer adds one to the address it just stored into and will automatically do the next store there (unless you put in a different address by pressing the "Store Address" key). Thus, to store into a series of consecutive memory locations you need only do Steps (1) and (2) above once, for the first memory location in the series. You then simply repeat Steps (3) and (4), typing in the numbers and pressing the "Store" key, as many times as necessary. The numbers will be entered into consecutive memory locations. Moreover, if you want to enter the same number into a series of locations you only need to do Step (3) once. The computer will remember the number you want to store, and each time you press the "Store" key it will store this same number in the next memory location.

As an illustration of this try the following exercise:

1. Type in F030

2. Press "Store Address"

3. Type in 55

4. Press "Store". F030 appears at the upper left, and one set of dots appears. 55 has been stored into location F030.

5. Press "Store" again. F031 appears at the upper left, and an identical set of dots appears right under the first set. 55 has been stored into location F031.

6. Type in AA

7. Press "Store". AA is stored into location F032. F032 appears at the upper left and a different pattern of dots appears under the first two sets.

8. Continue pressing "Store". AA is successively stored into F033, F034, F035, etc. When you have mastered this exercise, you know all there is to know about storing numbers into memory and are ready to learn how to fetch numbers back from memory.

2.3 FETCHING NUMBERS FROM MEMORY

Fetching numbers from memory works the same way as storing numbers to memory, using the two new keys shown in Figure 8,

X	X	X
X	X	X
X	X	X
FETCH	X	X

X	X	X
X	X	X
X	X	X
FET.AD.	X	X

Figure 8

the "Fetch Address" and "Fetch" keys. The sequence of steps to fetch a hex number from the computer's memory and to display it on the TV screen is as follows:

1. Type in the 4-digit hex number of the address in memory you want to fetch from. As usual, it appears at the upper right of the TV screen. (For example, type in F800.)

2. Press the "Fetch Address" key.

3. Press the "Fetch" key. This will cause the address of the memory location (in the example F800) to appear at the left of the TV screen, and will cause the two hex digits of information (either data or program) contained in that memory location to be fetched and displayed on the right side of the TV screen as the last two digits. (In the example the right side of the TV reads 0009; 09 is the contents of location F800. As when you were storing numbers to memory, the first two digits of the displayed number are irrelevant.)

You can examine a series of memory locations just by repeating Step (3). Each time you press the "Fetch" key, you will see the address on the left of the TV screen increase by 1, and two new digits will appear on the right of the TV indicating the contents of that memory location.

Exercise 3: Store 00, 01, 02, ..., 0F into the successive memory locations F300, F301, F302, ...F30F, and then fetch the numbers back out of those memory locations to insure that you stored them

correctly.

Answer:

1. Type in F300.
2. Press "Store Address".
3. Type in 00.
4. Press "Store". You should now see F300 0000 at the top of the TV.
5. Type in 01 and press "Store". You should see F301 0001.
6. Continue typing in numbers and pressing "Store" until you get to 0F. You should see F30F 0E0F on the TV.
7. Type in F300 and press "Fetch Address".
8. Press "Fetch". You should see F300 0000, verifying that the contents of memory location F300 are 00.
9. Press "Fetch" again. You should see F301 0001, showing that the contents of location F301 are 01.
10. Press "Fetch" again. You should see F302 0102, showing that the contents of location F302 are 02.
11. Continue pressing "Fetch", stepping through the memory addresses and verifying their contents until you get to F30F. The TV should display F30F 0E0F.

When you have mastered this exercise you will know how to store hex numbers (which can be either instructions or data) into the computer's memory, and how to fetch the numbers from memory to verify that they were stored correctly. The last thing you need to know before going on to write programs for the computer is the arrangement of the different areas of memory in your computer.

There are four distinct areas of memory:

1) Addresses F800-FFFF: ROM (read-only memory)

This area of memory can only be read (fetched from) and not stored to; hence, the name read-only memory. Its contents are permanent--you cannot change them, and more importantly, its contents remain the same even when you turn the power on and off. This type of memory is used in computer systems for storage of programs and data that are intended to be permanent and never change.

In your MagiCard system this area of memory contains a program called the monitor. This memory resides in the MagiCard on the large chip closest to the connector. The monitor program was permanently stored in this chip at our factory, and is instantly available to you whenever you turn on your MagiCard system.

This monitor program is the program that starts to run automatically whenever you turn your system on. You have been using this program all along as you have been learning to fetch and store. This monitor program reads the keyboard controllers and executes the commands you give it, as well as constantly updating the display on the TV screen. You will learn about other useful monitor functions shortly; in particular, you will learn how to tell the monitor program to transfer control from itself to a new program that you have written and stored in the computer's memory.

2) Addresses F000-F3FF: RAM (random-access memory)

This area of memory can be both fetched from and stored to. It is the area you have been using above in the store and fetch exercises. It is used for temporary storage of programs and data. The contents of RAM are lost when you turn the power off.

This RAM resides on your MagiCard on the two medium-sized chips next to the ROM chip. They contain a total of 1024 memory locations, each of which is large enough to contain two hexadecimal digits. This amount of memory is often referred to as 1K bytes of RAM: 1K meaning 1024 locations; byte meaning a location of memory big enough to contain two hex digits; and RAM meaning memory that can be both stored to and fetched from.

The various locations in the RAM memory can be used for whatever purposes you want, depending on what program is running at the time. For example, the monitor program uses locations F000 through F0D1 to save the data describing the TV display, which is why the display changed when you stored numbers into these locations. Eventually, you will want to store data into locations F000-F3FF from a program. Unlike other locations, this requires a special method described later in this chapter and in Section 7.2.

3) Addresses 0080-00FF: RAM

These addresses are for additional RAM located inside your Video Game Console. They are used in the same manner as the RAM described above, but without restrictions on use by a program.

4) Addresses 0000-003F: Video Display Generator

These addresses represent special locations in the video display generator chip inside your game console. Storing to these locations will change the display on your TV or produce sounds. You have already used two of these locations when you stored into locations 0008 and 0009 to change the color of the display. You will learn about a few of these locations as you examine some simple programs presented later in this chapter. More details are given in Chapter 6.

2.4 CREATING AND RUNNING PROGRAMS

You are now ready to learn to create programs for your computer, store them in memory, and run them. In this section you will go through the steps necessary to run programs, using the techniques to store and fetch you've already learned as well as the new keys "Shift" and "Run" shown in Figure 9.

RUN	X	X
X	X	X
HEX DUMP	INS DUMP	X
X	UN SHIFT	X

X	X	X
X	X	X
X	X	X
X	SHIFT	X

Figure 9

We will also describe the "Hex Dump" and "Ins Dump" (Instruction Dump) keys, which make it easier to verify that a program has been stored correctly in memory.

Any program, from the simplest to the most complex, must go through the same stages. First, the problem must be clearly stated. Next, the problem should be broken down into a series of small pieces. Finally, each of these pieces is translated into a series of computer instructions in computer machine language which can then be stored in the computer's memory. The program is then ready to run.

As a first simple example, we present a program to show all the different colors the video display generator can make on the TV. We go through the stages in writing this program in some detail to illustrate how a program is written. Later, examples will be more abbreviated, so you should spend some time on this first example to insure you thoroughly understand it.

First comes the statement of the problem. In our example, we want to display all colors on the TV. However, this is not really a precise enough statement of the problem. In general, it will be more useful to state the problem in a way more related to the computer. For this example, we need to recall just how to display a color on the TV: it is done by storing a number into memory location 0009. Thus, stated more precisely, the programming problem is to store all possible numbers in succession into location 0009.

Next, we break the problem up into pieces, at the same time deciding what registers or memory locations to use in the program for storage of data. In the example, there is only one piece of data that we are concerned with: the color we are about to display. We will use the X register of the computer to store this number. (The registers as well as the internal organization of the 6502 microprocessor are described in Chapter 3.) Thus the steps to solve this simple programming problem are:

A) Put 0 into the X register.

B) Store the X register into location 0009 (to start with color 0).

C) Add 1 to the X register (to go on to the next color).

D) Jump back to step (B) (so that steps B and C will be repeated to go through all the colors. Refer back to Chapter 1 if you have forgotten what a jump is.)

Note carefully how the program works. The first time Step B is executed there will be a value of zero in the X register, and thus color zero will be stored in location 0009 and displayed. The next time Step B is executed the X register will contain 1 (because 1 was added to the X register in Step C), and thus color 1 will be displayed. The next time around the X register will contain 2 (because another 1 was added to the X register the second time Step C was executed), and so color 2 will be displayed. The program will

step through all the numbers, and therefore all the colors. (In case you are wondering what happens when the X register counts up to FF (or 255 in decimal), the largest number that it can contain, it will return to zero and repeat the sequence of colors over again.)

This programming technique of executing the same instructions over and over again for different values of data (in this case for different numbers in the X register) is called a loop. Such loops are an extremely important programming tool. You will find them in virtually all the example programs and will find them to be very useful in writing your own programs. Three essential ingredients that are present in any loop are shown in their simplest forms in Steps B, C, and D of the example:

1. Do something (here Step B which displays a color).
2. Change the value of the data (here Step C which changes the value of the X register).
3. Go back to the start of the loop to do it again (here Step D which jumps back to Step B to repeat the loop).

Most loops also have a fourth element which is not present in this simple example: a check to see whether the program has gone through the loop enough times and should go on to something else. You will see this element of loops in future examples.

Finally, you are ready to translate each of the steps in the program into specific machine language instructions to the computer. In the simple example each step will correspond to one computer instruction. In more complicated examples, as you become more proficient in programming, each step will normally correspond to several computer instructions. The complete set of instructions that the 6502 computer can execute is given in Chapter 3. For now, you will learn the instructions a few at a time as you go through the sample programs. In each case the introduction of an instruction will

include the abbreviation for the instruction, the two digit hex code which is the actual machine language version of the instruction, and a brief description of what the instruction does. Again, for more details see Chapter 3.

LDX I A2 Load X register immediate: this instruction puts the two-digit hex number following A2 in the program into the X register.

STX 8E Store X register: this instruction puts the contents of the X register into the memory location whose 4-digit hex address follows 8E in the program.

INX E8 Increment X register: this instruction adds 1 to the X register.

JMP 4C Jump: this instruction jumps to the memory location whose address follows 4C in the program and continues executing instructions at the new memory location.

Using these instructions, the sample program is as follows (store the program in the computer starting at memory location F100, in the RAM, as indicated in the first column):

address	instruction	machine code	comment
F100	LDX I 00	A2 00	Put 00 into X register
F102	STX 0009	8E 09 00	Put X register into 0009
F105	INX	E8	Add 1 to X register
F106	JMP F102	4C 02 F1	Jump back to F102

A few points should be noted about this program. The first instruction is stored at location F100. The address of each subsequent instruction is determined by adding the length of the previous instruction to the address of the previous instruction. For example, the first instruction consists of the two bytes A2 and 00, which occupy memory locations F100 and F101. The second instruction goes in location F102. It is three bytes long (8E, 09, and 00) and so the third instruction will go in location F105, etc. You don't need to know all of these addresses to store the program in memory, since that is done just by repeated pressing of the "Store" key which automatically stores into successive memory locations. You do need to know the addresses of any instructions to which the program will jump

so that you can code the jump instructions. In the example, you need to know that the STX instruction is at location F102 so you can have the program jump to that instruction. Finally, note that when memory addresses (0009 and F102 in the example) are translated into machine code, the order of the bytes is reversed. For example, address F102 in the JMP instruction is entered as 02 first, and then F1. This is the way the 6502 expects to find its addresses.

You are now ready to store the program into memory and run it. As a reminder, the sequence of steps to store the program is:

1. Type in F100
2. Press "Store Address"
3. Type in A2, press "Store"
4. Type in succession 00,8E,09,00,E8,4C,02, and F1, pressing "Store" after each two-digit hex number. If you have done this correctly, when you are done the TV display should read F108 02F1 .

Before running the program, it is a good idea to check the contents of memory to verify that the program was stored correctly. You could do this by fetching bytes from memory one at a time as described above, but there are two easier ways to do this by using the "Hex Dump" and "Ins Dump" keys. To use the Hex Dump feature, use the following sequence:

1. Type in F100
2. Press "Fetch Address"
3. Press the "Shift" key (0 on the right controller)
4. Press the "Hex Dump" key (on the left controller). You will see an address (F100) together with two bytes of data (A2 00) on the bottom line of the TV. The two bytes are the contents of F100 and F101.
5. Press "Hex Dump" again. You will see a new line with an address (F102) and the contents of the next two memory locations (8E 09).

6. Continue pressing "Hex Dump" to examine the remainder of the program. When you have gone through the entire program, be sure to press "Unshift" (the 0 key on the left controller) to return to normal mode. You will know you are in normal mode because the cursor, the little square in the top center of the TV, will begin blinking again.

To use the Instruction Dump feature, use the same sequence but press the "Ins Dump" key (key 8 on the left controller) instead of the "Hex Dump" key. Now you will see the abbreviation for the actual instruction, just as you coded the program above. (Again make sure to press "Unshift" when you are finished looking at the whole program.)

Finally, you are ready to run the program! The sequence of steps is as follows:

1. Type in F100
2. Press "Store Address" key
3. Press "Shift" key
4. Press "Run" key (1 key on left controller). At this point the MagiCard gives you a chance to stop the run sequence to prevent you from inadvertently running a program before you are ready. The screen will go blank and wait for you to press either the "Game Reset" or "Game Select" switches on the game console. Pressing the "Game Reset" switch will return you to the monitor, in case you were not really ready to run the program. However, here you are ready to run, so
5. Press the "Game Select" switch to begin execution of the program.

You should now see a brilliant display of colors on your TV. Congratulations, you have run your first program!

What next? You'll probably get tired of watching the colors fairly soon and will want to go on to bigger and better programs, but there are still a few things you can learn from this simple example. For one thing, how can you stop the program and get back to the monitor? The answer is, you can't! The program does not contain any instructions to stop, and so there is no way for it to stop. It will just keep running, showing the colors over and over again, until you

turn your computer off. In the future, it will be useful to put instructions into your programs to allow them to stop, so this is the first modification to make to the sample program. As part of the loop, check the "Game Reset" switch, and if it is pressed leave the color display program and go back to the monitor program. The modified program looks like this:

```
F100    LDX I 00      A2 00
F102    STX   0009    8E 09 00
F105    INX           E8
F106    LDA   0282    AD 82 02   These four instructions test the
F109    AND I 01      29 01      game reset switch and jump back to
F10B    BNE   01      D0 01      the monitor (via the BRK instruction)
F10D    BRK           00         if game reset is pressed
F10E    JMP   F102    4C 02 F1
```

Try typing this in and running it. (You'll have to turn the computer off and then back on again to get the original example program to stop.) You will again see the display of colors, but now, if you press the "Game Reset" switch while the program is running, the program will jump back to the monitor program (I'll explain how this works in a moment). The monitor will as usual identify itself with eMx-2 in the middle of the TV and with the blinking cursor at the top. To restart the program, just key in F100, press "Store Address", "Shift", "Run", and "Game Select", and you should see the display of colors again. To return to the monitor again, press "Game Reset". You should be able to go back and forth between the color display program and the monitor as many times as you like.

Now think about what it is that this new version of the program did. The new instructions you used (again see Chapter 3 for detailed descriptions) are:

LDA AD Load Accumulator: The accumulator is another one of the registers in the 6502 CPU, like the X register. This instruction puts the contents of the memory location whose address follows AD into the accumulator. In the example, the address is 0282.

AND I 29 And Immediate: This instruction takes the logical and of the immediate byte with the accumulator and puts the result back in the accumulator. The immediate byte is the byte following 29.

BNE D0 Branch if not equal: This is the first example of a conditional instruction; that is, an instruction that does one of two different things depending on some condition. In this case, the condition is whether or not the last instruction produced a result equal to zero. If it did, this instruction does nothing and the program continues with the next instruction in sequence after the BNE instruction. If the result of the previous instruction was not equal to zero, a branch occurs. A branch is like a jump, causing the program to jump off to a location different than the one next in sequence. Branches differ from jumps in that they tell the program how far to jump rather than where to jump, as JMP instructions do. In the example, the byte after D0 in the program is 01, meaning that if the condition for the branch is satisfied (here if there was a not zero result from the last instruction), it jumps 01 bytes; that is, it jumps past the BRK instruction and executes the JMP F102 instruction instead. To summarize:

if the last result was:	then:	and next is:
equal	no branch	BRK
not equal	branch	JMP F102

BRK 00 Break: This instruction causes the program to stop whatever it is doing and automatically jump to a predetermined location. In your MagiCard system this location is the monitor program. Thus, executing a break instruction is a quick and easy way of getting back to the monitor.

Now you can see how this new program works. The LDA 0282 instruction puts the contents of memory location 0282 into the accumulator. This is a special location in the input/output section of your computer (see Chapter 6 for a detailed explanation) which contains the values for the various switches on the game console. The "AND I 01" instruction selects the first bit only (which corresponds to the "Game Reset" switch) for testing. (If you don't know what a bit is, see below.) The BNE instruction then does different things depending on whether or not you press the "Game Reset" switch. If you do not press the switch, you will get a not equal to zero result from the AND instruction. Thus the program will take the branch, skip the BRK instruction, and continue to execute the color display program.

On the other hand, if you have pressed the switch, you will get an equal to zero result from the AND. The program will not take the branch and thus will execute the BRK instruction and return to the monitor. (By the way, you can check the position of the other switches in the same manner by using "AND I 02" for the "Game Select" switch, "AND I 80" for the "Right Difficulty" switch, "AND I 40" for the "Left Difficulty" switch, and "AND I 08" for the "TV Type" switch. In all cases you get a not equal to zero result from the AND if the switch is up and an equal to zero result if the switch is down.)

Such conditional branches as used in this example are another extremely important programming tool. They provide the way a program can make decisions and alter its actions, based either on an external event like the setting of a switch or on the result of an internal computation. For example, conditional branches are the normal way for loops to check on whether or not they should be executed again. You will see conditional branches used repeatedly in the sample programs in the next section.

Finally, a note about bits and bytes. As you probably know, computers do not actually use hexadecimal numbers, but rather binary (or base-2) numbers, where the only digits are zero and one. This is a particularly convenient number system for computers because zero and one can correspond to the "off" and "on" states of an electronic circuit. All information in a computer is nothing more than a long string of "ons" and "offs", ones and zeros. Hexadecimal is a useful shorthand for programmers because of the close correspondence between hex and binary numbers, as can be seen from the following chart of the binary (and decimal) equivalents of the hex digits:

decimal	hex	binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

You can see that each hex number corresponds to a unique four-digit binary number. Each hex digit is four binary digits, or bits (bit being an abbreviation for binary digit). A byte, or two hex digits, is 8 bits. Thus, each location in the computer's memory is actually a collection of 8 little electronic circuits, each of which can be individually turned on and off, corresponding to 8 bits or 1 byte. As the 8 bits take on all possible values of on and off, they produce all possible combinations of two hex digits (all the hex numbers from 00 to FF, or all the decimal numbers from zero to 255).

For example, the decimal number 171 in hexadecimal is AB (see the chart in Appendix A), and the binary equivalent of AB is 1010 1011 (see chart above). Thus decimal 171, or hex AB, is represented in the computer as a byte with bits 0,1,3,5, and 7 on (starting from zero and counting from the right), and bits 2,4, and 6 off. Bits and bytes will be referred to in Chapter 3 in the discussion of the 6502 microprocessor.

2.5 USING MONITOR SUBROUTINES

You are now ready to write more complex sample programs using another important programming tool, subroutines. A subroutine is simply a section of your program that you need to execute fairly often. Rather than repeat the same instructions over and over again at different places in your program, you put the instructions in one place only. At all the places where you want to execute that whole set of instructions, you put the single instruction JSR (Jump to Subroutine) with the address of the one place where the set of instructions begins. This will cause the program to jump off and execute the set of instructions (the subroutine). At the end of the subroutine a special instruction, RTS (Return from Subroutine), forces a jump back to the instruction following the JSR, where execution continues.

A number of especially useful subroutines are contained in the monitor program and are available to the user. They are fully described in Chapter 7. A few of them are used in the following sample programs to illustrate display of both text and graphics on the TV screen. A complete description of all the instructions used in these programs will not be given here, since full details on all 6502 instructions are given in Chapter 3. Instead full listings of the programs will be given, ready to be keyed in and/or modified to suit your own purposes. Descriptions of what the programs do, and comments on various parts of the programs where new things are introduced are also provided.

Example 1 displays the full MagiCard character set on your TV screen using the monitor subroutines DOLN and DSPL. As explained in Appendix B, each hex number from 00 to 3F represents a unique character. This program displays all the characters by putting all

these hex numbers into the display area of memory (locations 98-A1). This is first done in the loop starting at location LOOP, where the numbers from 00 to 09 are stored into locations 98 to A1. Note the use of the X register as an index register in the "STA X 98" instruction, which stores into a different memory location for each value of the X register. As the X register takes on all values from 00 to 09, these numbers are successively stored into 98, 99, 9A, etc. as the program goes through the loop. The loop ends when the X register reaches 0A, and the "JSR DOLN" instruction jumps to subroutine DOLN which displays the first line of characters. Location 8F is then increased by one to point to the next line of the display, and locations 98 through A1 each have 0A added to them (in the loop starting at location LOOP2, again using indexed addressing mode), to move on to the next set of ten characters (in this case characters 0A through 13). This process continues until all 7 lines have displayed, whereupon the program jumps to location DONE. Here it just continuously calls DSPL to continue showing the same display on the TV until you hit the "Game Reset" switch, which will return you to the monitor. (You don't need to put code to check for the "Game Reset" switch into your program because this is automatically done by the DSPL subroutine.)

Example 2 is a modification of the first program which allows you to type messages on the TV screen using the keyboard controllers. Keys 1-9 on the left controller represent letters A-I; keys 1-9 on the right controller represent J-R; pressing key 0 on the right controller puts you in shift mode, whereupon keys 1-9 on the left controller represent S-Z and blank. Key 0 on the left controller takes you out of shift mode again. This program again uses DSPL to put up the display and also to read the keyboard controllers, and uses subroutine ONEC to put one character at a time onto the display as you

enter them on the keys. Note the use of certain memory locations to save data during the course of the program; location E0 contains the line number of the character being displayed, E1 contains the position within the line, and E2 is a flag remembering whether or not you are in shift mode. E2 contains 0 when in unshift mode and 12 when in shift mode. Studying these two examples should enable you to write your own programs to display text on the TV screen.

Example 3 shows how to use the CALP subroutine in conjunction with STPM and DSPL to produce graphic displays. It is a target practice game using the left controller only. Key 4 moves your cannon to the left, key 6 moves your cannon to the right, and key 0 fires a bullet at the moving targets. Note the use of subroutines within the main program (OFF to turn a point off and ON to turn a point on), and the use of the sound registers in the video display generator to make noises when a bullet is fired and when a hit is made. Also note the extensive use of memory locations to store variables (locations E0-EC), as explained in the comments in the program listing. You can easily modify this program to add multiple targets, to keep score at the top of the screen using the text display routines, and to use both controllers to allow for two different players, each having a cannon and competing for high score.

Finally, Example 4 shows you how to run the TV display without using monitor subroutines. It makes all the appropriate stores to locations in the video display generator to synchronize the TV picture. A program can make changes to the various display registers of the display generator in the course of execution to change the picture being displayed. In the example this is done in two places; during LOOP3, the pattern is changed in location 0E and the right-side color is changed in location 07 between lines of a single TV picture; and in the major loop of the program running from START to LOOP2, the

left-side color is changed in location 06 and the pattern is changed in location 0F between frames of the TV picture. You can easily see these two types of changes as you watch the TV: the right-side color, for example, changes each line of the display; while the left-side color only changes from one TV picture to the next. By making modifications of this program, you will be able to make displays of various types of objects that move around on the TV screen and begin to duplicate the video effects of game cartridges.

Example 1: Display of Character Set

```
START   LDA I 00     F100   A9 00       0 to accumulator
        STA  008F            8D 8F 00    0 to 8F (line of display)
        LDX I 00             A2 00       0 to X reg. (character position)
LOOP    TXA          F107    8A          transfer X reg to acc.
        STA ZX 98            95 98       store acc. in display area of mem.
        INX                  E8          increment X reg.
        CPX I 0A             E0 0A       compare X reg with 0A (to see if
                                         line done)
        BNE LOOP             D0 F8       branch back to loop if not done
DISP    JSR DOLN     F10F    20 C8 FC    jump to display line subroutine
        LDX 008F             AE 8F 00    put line # in X reg
        INX                  E8          add 1 to X reg
        STX 008F             8E 8F 00    put new line # back in 8F
        CPX I 07             E0 07       check if all lines finished
        BEQ DONE             F0 11       jump to done if all lines finished
        LDX I 00             A2 00       0 to X (character position)
        CLC                  18          clear carry (for add instruction)
LOOP2   LDA ZX 98    F120    B5 98       get old character to accumulator
        ADC I 0A             69 0A       add A to character to make new one
        STA ZX 98            95 98       store new character in display mem.
        INX                  E8
        CPX I 0A             E0 0A       check if done with line
        BNE LOOP2            D0 F5       branch back to loop2 if not done
        JMP DISP             4C 0F F1    jump to disp if done
DONE    JSR DSPL     F12E    20 6F FA    jump to dspl subroutine
        JMP DONE             4C 2E F1    jump back to done to repeat
                                         display until game reset is hit
```

Example 2: Memo Pad

```
START   LDX I 00     F100   A2 00      0 to X reg
        STX Z E0             86 E0     0 to E0 (line #)
        STX Z E1             86 E1     0 to E1 (position in line)
        STX Z E2             86 E2     0 to E2 (shift flag)
DISP    JSR DSPL     F108   20 6F FA   jump to dspl subr. to display
                                       a line and read keyboard controllers
        LDA Z 83             A5 83     right keyboard byte to acc.
        BPL RIGHT            10 44     branch to right if key was hit
        LDA Z 82             A5 82     left keyboard byte to acc.
        BMI DISP             30 F5     branch back to disp if no key was hit
LEFT    BNE LCHAR    F113    D0 07     branch to lchar if char key was hit
UNSHFT  LDX I 00     F115    A2 00     unshift key was hit; 0 to X reg
        STX Z E2             86 E2     0 to E2
        JMP DISP             4C 08 F1  jump to disp
LCHAR   CLC          F11C    18        char key was hit; clear carry
        ADC E2               65 E2     add shift flag (0 or 12) to char
        CMP I 1B             C9 1B     check if blank character
        BNE NTBLNK           D0 02     branch if not blank
        LDA I 20             A9 20     put code for blank (20) in acc.
NTBLNK  STA Z E3     F125    85 E3     acc to E3 (new character to E3)
NEWCHR  LDX Z E0     F127    A6 E0     line # to X reg
        STX Z 8F             86 8F     line # to 8F
        LDX Z E1             A6 E1     position in line to X reg
        STX Z 8E             86 8E     position in line to 8E
        LDA Z E3             A5 E3     character to accumulator
        STA ZX 98            95 98     store character in right location in
                                       memory for one character display subr.
        JSR ONEC             20 67 FB  jump to one char subroutine
        LDX Z E1             A6 E1     position in line to X reg
        INX                  E8        add 1 to X reg
        STX Z E1             86 E1     new position to E1
        CPX I 0A             E0 0A     check if at end of line
        BNE DISP             D0 C9     branch back to disp if not end of line
        LDX I 00             A2 00     0 to x reg
        STX Z E1             86 E1     0 to E1 (start of new line)
        LDX Z E0             A6 E0     line # to X reg
        INX                  E8        add 1 to X
        STX Z E0             86 E0     new line # to E0
        CPX I 07             E0 07     check for last line on screen
        BNE DISP             D0 BC     jump back to disp if not last line
        LDX I 00             A2 00     0 to X reg
        STX Z E0             86 E0     0 to E0 (restart with line # 0)
        JMP DISP             4C 08 F1  jump back to disp to continue
RIGHT   BNE RCHAR    F153    D0 07     check for new char on right controller
SHFT    LDX I 12             A2 12     shift key was hit; 12 to X reg
        STX Z E2             86 E2     store 12 in E2 (to shift characters)
        JMP DISP             4C 08 F1  jump back to disp
RCHAR   CLC          F15C    18        new character on right; clear carry
        ADC I 09             69 09     add 9 to acc to convert char
        STA Z E3             85 E3     store new char in E3
        JMP NEWCHR           4C 21 F1  jump to newchr to put up new char
```

Example 3: Target Practice

```
START   LDX I F4     F100   A2 F4      set up 8C and 8D for stpm
        STX Z 8D            86 8D
        LDY I 00            A0 00
        TYA                 98
        STY Z 8C            84 8C
LOOP1   JSR STPM     F109   20 F7 FF   loop to clear TV display
        INY                 C8         by storing 00 in F100-F1D2
        CMY I D2            C0 D2      using stpm subroutine
        BNE LOOP1           D0 F8
        STA Z E3     F111   85 E3      initialize variables; 0 to E3
        STA Z EA            85 EA      0 to EA  (EA and EB are address for
        LDA I F8            A9 F8                random number generated by
        STA Z EB            85 EB      F8 to EB reading a loc in monitor)
        LDA I 14            A9 14      14 to E0 (starting horizontal pos
        STA Z E0            85 E0                of cannon)
        LDA I 02            A9 02      02 to E7  (speed of bullet)
        STA Z E7            85 E7
        LDA I 28            A9 28      28 is vertical pos of cannon
        STA Z 8F            85 8F      turn on cannon by putting hor.
        LDA Z E0            A5 E0      and vert. position in 8E and 8F
        STA Z 8E            85 8E      and calling subroutine on
        JSR ON              20 15 F2
NEWTRG  LDX I 00     F12C   A2 00      new target; 00 to E4 (hor. pos.)
        STX Z E4            86 E4
        LDA X) EA           A1 EA      get a random instruction from mon.
        INC Z EA            E6 EA
        TAX                 AA         save acc in X reg
        AND I 0F            29 0F      calculate target v position
        ADC I 10            69 10
        STA Z E5            85 E5      store targ v position
        TXA                 8A         get acc back from x reg
        ROR                 6A         shift right four bits
        ROR                 6A
        ROR                 6A
        ROR                 6A
        AND I 03            29 03      calculate target speed
        ADC I 01            69 01
        STA Z E6            85 E6      target speed to E6
        STA Z E8            85 E8       and E8 (target speed counter)
DISP    JSR DSPL     F148   20 6F FA   jump to display subr.
        DEC Z EC            C6 EC      check sound counter
        BPL ARND            10 04
        LDA I 00            A9 00      turn off sound
        STA Z 16            85 16
ARND    LDA Z 8A     F153   A5 8A      read left keyboard controller
        BMI NOKEY           30 52       and jump to nokey if no key hit
        BEQ FIRE            F0 34      jump to fire if 0 key hit
        CMP I 05            C9 05      check whether left (key 4) or
        BPL RIGHT           10 17       right (key 6) movement wanted
LEFT    LDA Z E0     F15D   A5 E0      get cannon horizontal pos.
        BEQ NOKEY           F0 48       and jump to nokey if at left edge
        STA Z 8E            85 8E
        LDA I 28            A9 28
        STA Z 8F            85 8F
        JSR OFF             20 1F F2   turn off old cannon position
        DEC Z E0            C6 E0      move cannon one pos. left
        DEC Z 8E            C6 8E
```

```
        JSR ON          20 15 F2  turn on new cannon position
        JMP NOKEY       4C A9 F1  jump to nokey to continue
RIGHT   LDA Z E0  F174  A5 E0     get cannon hor. pos.
        CMP I 27        C9 27     check if at right edge
        BEQ NOKEY       F0 2F      and jump to nokey if yes
        STA Z 8E        85 8E
        LDA I 28        A9 28
        STA Z 8F        85 8F
        JSR OFF         20 1F F2  turn off old cannon position
        INC Z E0        E6 E0     move cannon one pos. right
        INC Z 8E        E6 8E
        JSR ON          20 15 F2  turn on new cannon position
        JMP NOKEY       4C A9 F1  jump to nokey to continue
FIRE    LDA Z E3  F18D  A5 E3     check if bullet already fired
        BNE NOKEY       D0 18      and jump to nokey if yes
        INC Z E3        E6 E3     turn on bullet fired flag
        LDA I 27        A9 27     starting bullet vert. pos.
        STA Z E2        85 E2     bullet vert. pos. to E2
        STA Z 8F        85 8F
        LDA Z E0        A5 E0     get cannon hor. pos. for
        STA Z E1        85 E1      starting bullet hor. pos.
        STA Z 8E        85 8E
        JSR ON          20 15 F2  turn on bullet
        LDA Z E7        A5 E7     get bullet speed to
        STA Z E9        85 E9      bullet speed counter
        JSR BSND        20 2B F2  make bullet sound
NOKEY   LDA Z E3  F1A9  A5 E3     check if bullet fired
        BEQ NOBULL      F0 23      and jump to nobull if not
        DEC Z E9        C6 E9     decrement bullet speed counter
        BNE NOBULL      D0 1F      and jump if not time to move
        LDA Z E7        A5 E7     restore bullet speed counter
        STA Z E9        85 E9
        LDA Z E1        A5 E1     bullet hor. pos.
        STA Z 8E        85 8E
        LDA Z E2        A5 E2     bullet vert. pos.
        STA Z 8F        85 8F
        JSR OFF         20 1F F2  turn off old bullet position
        DEC Z E2        C6 E2     move bullet up one position
        BMI TOP         30 08     jump to top if at top of screen
        DEC Z 8F        C6 8F
        JSR ON          20 15 F2  turn on new bullet position
        JMP NOBULL      4C D0 F1
TOP     LDA I 00  F1CC  A9 00     turn off bullet fired flag
        STA Z E3        85 E3      by putting 00 into E3
NOBULL  DEC Z E8  F1D0  C6 E8     decrement target speed counter
        BNE NOMOVE      D0 1D      and jump if not time to move
        LDA Z E6        A5 E6     restore target speed counter
        STA Z E8        85 E8
        LDA Z E4        A5 E4     target hor. pos.
        STA Z 8E        85 8E
        LDA Z E5        A5 E5     target vert. pos.
        STA Z 8F        85 8F
        JSR OFF         20 1F F2  turn off old target position
        LDX Z E4        A6 E4
        CPX I 27        E0 27     check if target at right edge
        BEQ JUMPNT      F0 26      and jump to newtrg if yes
        INX             E8        move target one position right
        STX Z E4        86 E4
        STX Z 8E        86 8E
        JSR ON          20 15 F2  turn on new target position
NOMOVE  LDA Z E1  F1F1  A5 E1     compare target and bullet
```

```
        CMP Z E4          C5 E4      horizontal positions
        BNE JMPDSP        D0 1B      and jump to disp if different
        LDA Z E2          A5 E2     compare target and bullet
        CMP Z E5          C5 E5      vertical positions
        BNE JMPDSP        D0 15      and jump to disp if different
        JSR HITSND        20 3A F2  hit!! make hit sound
        LDA Z E1          A5 E1
        STA Z 8E          85 8E
        LDA Z E2          A5 E2
        STA Z 8F          85 8F
        JSR OFF           20 1F F2  turn of bullet and target
        LDA I 00          A9 00
        STA Z E3          85 E3     turn off bullet fired flag
JUMPNT  JMP NEWTRG  F20F  4C 2C F1  jump to newtrg
JMPDSP  JMP DISP    F212  4C 48 F1  jump to disp
ON      JSR CALP    F215  20 BE FF  on subr; jump to calp
        ORA Y F000        19 00 F0  or new bit on
        JSR STPM          20 F7 FF  jump to stpm
        RTS               60        end of subroutine; return
OFF     JSR CALP    F21F  20 BE FF  off subr; jump to calp
        EOR I FF          49 FF
        AND Y F000        39 00 F0  and new bit off
        JSR STPM          20 F7 FF  jump to stpm
        RTS               60        end of subroutine; return
BSND    LDA I FF    F22B  A9 FF     bullet sound subroutine
        STA Z 1A          85 1A     load sound registers
        LDA I 22          A9 22
        STA Z 18          85 18
        STA Z 16          85 16
        LDA I 20          A9 20
        STA Z EC          85 EC     set up sound counter (EC)
        RTS               60        end of subr, return
HITSND  LDA I 77    F23A  A9 77     hit sound subroutine
        STA Z 16          85 16     load sound register
        LDA I 20          A9 20
        STA Z EC          85 EC     set up sound counter
        RTS               60        end of subr, return
```

Example 4: Generating Your Own Display

```
START   LDA Z 81    F200  A5 81       contents of 81 to acc
        STA Z 0F          85 0F       save new pattern in 0F
        LDA I 03          A9 03       3 to acc
        STA Z 0A          85 0A       3 to 0A
        LDA I 55          A9 55       55 to acc.
        STA Z 07          85 07       55 to 07 (right side color)
        LDY I 00          A0 00       0 to Y reg
        DEY               88          decrement y reg
        STA Z 02          85 02
        STY Z 01          84 01       start V blank
        STY Z 00          84 00       start V sync
        LDA I 2A          A9 2A       2A to acc
        STA 0295          8D 95 02    store 2A into timer
LOOP1   LDY 0284    F21A  AC 84 02    load timer to y and wait
        BNE LOOP1         D0 FB       for it to run out
        STY Z 02          84 02
        STY Z 00          84 00       end V sync
        LDA I 24          A9 24       24 to acc
        STA 0296          8D 96 02    load timer
        LDA 0282          AD 82 02    switches to acc
        AND I 01          29 01       look at game reset switch
        BNE NORSET        D0 01       branch if switch not hit
        BRK               00          break to monitor if reset hit
NORSET  INC Z 80    F230  E6 80       increment location 80
        BNE LOOP2         D0 0A
        LDA I E0          A9 E0       E0 to acc
        STA Z 80          85 80       E0 to 80 to reset counter
        INC Z 81          E6 81       increment loc 81 (color)
        LDA Z 81          A5 81       new color to acc
        STA Z 06          85 06       new color to 06 (left side color)
LOOP2   LDY 0284    F23E  AC 84 02    wait for timer
        BNE LOOP2         D0 FB
        STY Z 02          84 02       start of TV scan line
        STY Z 01          84 01       end V blank
        LDX I E4          A2 E4       E4 to X reg
LOOP3   STY Z 02    F249  84 02       start of TV scan line
        STX Z 0E          86 0E       new pattern to 0E
        STX Z 07          86 07       new color to 07 (right side color)
        DEX               CA          decrement X reg
        BNE LOOP3         D0 F7       jump back to loop3 for
                                      next line of TV picture
        JMP START         4C 00 F2    jump back to start
```

CHAPTER 3

Description of 6502 Microprocessor

This chapter gives a complete description of the 6502 microprocessor. It is assumed that those who read this chapter have a good general understanding of computer concepts (such as bytes, bits, hexadecimal, registers, memory, and instructions). Although an effort has been made to make this MagiCard manual as complete as possible, you may find it necessary to read some additional material on microcomputers to fully understand the material in this chapter. You should be able to find several books on the 6502 in any small computer store. An excellent example is the book Programming The 6502, third edition by Rodnay Zaks.

3.1 General Description

The 6502 microprocessor is a general purpose 8 bit microprocessor. The processor contains six registers--one accumulator, two index registers, a stack pointer, a processor-status register, and a program counter. A maximum of 65,536 bytes of memory can be used with the 6502 although the design of the ATARI Video Computer System limits the size of a MagiCard program to about 1024 bytes. The 6502 has an instruction set of sixty-five instructions and thirteen addressing modes (many of which can be used with only a few instructions). All 6502 instructions are either one, two, or three bytes in length. The first byte of the instruction (the "opcode") specifies the operation

to be performed while the second and third bytes (if present) provide either data for the instruction or the address of a memory location to be referenced by the instruction.

Section 3.2 describes the 6502 registers. Section 3.3 describes the format of 6502 instructions including a detailed description of the possible addressing modes. Finally, Section 3.4 describes the operation of each of the 6502 instructions.

3.2 Description of 6502 Registers

The names, abbreviations, and lengths of the 6502 registers are given below:

Register Name	Abbreviation	Length
Accumulator	A	8 bits
X index register	X	8 bits
Y index register	Y	8 bits
Stack pointer register	S	8 bits
Processor status register	P	8 bits
Program Counter	PC	16 bits

The use of each of the registers will now be discussed in more detail.

3.2.1 Accumulator

The accumulator is used primarily to perform arithmetic and logical operations on data.

3.2.2 X and Y Index Registers

The index registers are used to specify an offset (index) into a table of data and as counters.

3.2.3 Stack Pointer Register

The stack pointer contains the address of the next memory location available on the "stack". The stack is an area of memory (confined on your ATARI game to addresses between 80 and FF) that is used for the temporary storage of data. In particular, the stack is used as part of the 6502 subroutine calling procedure (see description of JSR and RTS instructions in Section 3.4.11) and the 6502 "break" mechanism (see description of the BRK instruction in Section 3.4.15).

Bytes of data may be placed on the stack ("pushed") one at a time and, at some future time, removed from the stack ("pulled") one at a time in the opposite order in which they were pushed. Pushing a byte onto the stack causes the byte to be stored in memory at the address contained in the stack pointer, after which the stack pointer is decremented by one. Pulling a byte from the stack is the inverse of pushing--the stack pointer is incremented by one, and the address now in the stack pointer is taken as the address of the byte to be removed. Because of the way the "push" and "pull" operations are defined, the stack grows from larger addresses to smaller ones. For this reason, the stack pointer is usually set to an initial value of FF.

3.2.4 Processor-Status Register

The processor-status register contains a collection of status flags. Each flag has two possible values--a value of one (indicating that the flag is "set") or a value of zero (indicating the the flag is "clear"). Depending on the particular flag, the value of a flag is either altered by including a special instruction in the program, or altered automatically by the microprocessor to indicate something about the outcome of the instruction it has just executed. The values

of some of the flags can be altered in both ways.

Those flags that are automatically altered by the microprocessor are called "condition codes". Depending on the particular instruction that has been executed, the microprocessor may alter the value of any or all of the condition codes. Those condition codes that are altered by the execution of an instruction are specified in the detailed description of each instruction given in Section 3.4.

The names of the flags and a short description of their use are given below.

Zero Flag (or "Z-flag")-- This flag is a condition code set by the microprocessor to indicate that a byte had a value of zero.

Negative Flag (or "N-flag")-- This flag is a condition code set by the microprocessor to indicate that a byte had a negative value (i.e., its most significant bit was a one).

Carry Flag (or "C-flag")-- This flag is a condition code set by the microprocessor to indicate that a logical (e.g., a shift) or arithmetic (e.g., an addition) operation was performed that resulted in a value greater than could be stored in a one byte number.

Overflow Flag (or "V-flag")-- This flag is a condition code set by the microprocessor to indicate that the result of an addition or subtraction operation resulted in "arithmetic overflow". Arithmetic overflow occurs in an addition operation when two numbers with the same sign are added together and the result is a number with the opposite sign. For example, if you add the number one to the number 7F (hex), the result is 80 (hex). Both the number one and the number 7f are positive (the most significant bit in the byte is zero) while the number 80 is negative (the most significant bit in the byte is one). Thus, arithmetic overflow has occurred. In a similar fashion, arithmetic overflow occurs in a subtraction operation when two numbers of opposite sign are subtracted and the sign of the result is the same as the sign of the number subtracted.

Decimal-Mode Flag (or "D-flag")-- When this flag is set by the programmer, it tells the microprocessor to perform add and subtract operations in "packed decimal" mode. In packed decimal mode, each byte of data is treated as if it contained two decimal digits--the most significant digit in the most significant four bits of the byte, and the least significant digit in the least significant four bits of the byte.

Interrupt-Disable Flag (or "I-flag")-- When this flag is set by

the programmer, it tells the microprocessor that the BRK instruction (see Section 3.4.15) is to be executed in the same way as the NOP instruction (see Section 3.4.13)--i.e. the BRK instruction is essentially ignored.

Break Flag (or "B-flag")-- This flag is set by the microprocessor whenever the execution of a BRK instruction has caused an interrupt (see Section 3.4.15) and is of essentially no value to users of the MagiCard.

Each flag occupies one bit in the processor status register. The particular bit position of each flag is of very little interest to the programmer but is included here for completeness:

Bit	Flag
0	Carry Flag
1	Zero Flag
2	Interrupt-Disable Flag
3	Decimal Mode Flag
4	Break Flag
5	unused
6	Overflow Flag
7	Negative Flag

3.2.5 Program Counter

The program counter contains the 16-bit address of the next instruction to be executed by the microprocessor. It is automatically updated after the execution of each instruction.

3.3 Format of 6502 instructions

All 6502 instructions consist of one, two, or three bytes. The first byte of the instruction (called the "opcode") tells the 6502

1. What operation is to be performed.
2. How may bytes are in the instruction (i.e. one, two, or three).
3. What the additional bytes in the instruction are to be used for.

The possible operations that can be performed by 6502 instructions will be given when the instruction set is described in Section 3.4. The second and third items in this list are collectively referred to as the "addressing mode" and will now be discussed in more detail.

The 6502 microprocessor recognizes thirteen different addressing modes. Two of the addressing modes tell the 6502 that the only byte in the instruction is the opcode. One of the addressing modes tells the 6502 that there are two bytes in the instruction and that the second byte contains the actual data to be used in performing the requested operation. For all other addressing modes, the one or two bytes following the opcode are used by the 6502 to calculate a "target address" for the instruction.

The exact manner in which the target address is calculated depends on the particular addressing mode selected. A description of the target address calculation is included as part of the description of each addressing mode given in Sections 3.3.1 to 3.3.12. The use that is made of the target address depends on the operation being performed. For example:

If the instruction is a register load operation, the target address specifies the address in memory whose contents is to be loaded into the register.

If the instruction is a register store operation, the target specifies the address in memory into which the value in the register is to be stored.

If the instruction is a jump operation, the target address specifies to what address the microprocessor is to jump.

Many operations can be combined with more than one addressing mode. For example, consider the operation that shifts a byte of data one bit to the left. Using one of the possible 6502 addressing modes for this instruction, you can specify the 16-bit address of the data byte that is to be shifted. Alternatively, by specifying a different

addressing mode, you can shift a data byte that is in the accumulator.

It is important to note that not all operations can be combined with all addressing modes (i.e., no opcode exists for many imaginable combinations of operations and addressing modes). Until you are familiar with the 6502, you will have to take care to insure that a program you are writing does not depend crucially on a very reasonable (but nonexistent!) operation and addressing mode combination.

All the addressing modes used by the 6502 are described below. The abbreviations for the addressing modes used by the MagiCard instruction dump feature (see Section 4.3) are shown within square brackets.

3.3.1 [ ] "Accumulator" addressing mode or "Implied" addressing mode:
Total instruction length is one byte. Accumulator addressing means that the accumulator contains the operand (data to be used by the instruction). Implied addressing means that the operand is implied by the operation itself.

3.3.2 [I] "Immediate" addressing mode:
Total instruction length is two bytes. The second byte of the instruction is the actual value of the operand. Another way to think of this is that the target address for the operation is the address of the second byte of the instruction.

3.3.3 [A] "Absolute" addressing mode:
Total instruction length is three bytes. The second and third bytes of the instruction contain the target address for the operation. The second byte of the instruction contains the least significant eight

bits of the address, and the third byte of the instruction contains the most significant eight bits of the address.

3.3.4 [Z] "Zero Page" addressing mode:

Total instruction length is two bytes. The second byte of the instruction contains the target address. This mode is similar to the absolute addressing mode except only addresses from 00 to FF may be referenced.

3.3.5 [ZX] "Zero Page X Indexed" addressing mode:

Total instruction length is two bytes. The target address is found by adding the second byte of the instruction to the contents of the X index register (only the lower eight bits of the sum are kept).

3.3.6 [ZY] "Zero Page Y Indexed" addressing mode:

The same as zero page X indexed except the Y index register is used. Note: this mode is available for only two instructions--load X index register (LDX) and store X index register (STX).

3.3.7 [X] "Absolute X Indexed" addressing mode:

Total instruction length is three bytes. The second and third bytes of the instruction contain a sixteen-bit address in the same manner as in absolute addressing. The target address is found by adding the contents of the X index register to this sixteen-bit address. (Note: The addition of the X index register to the original sixteen-bit address is performed in a sixteen-bit manner. For example, if the second and third bytes of the instruction were C9 and FA, and the

contents of the X index register were 40, then the target address would be FAC9 + 40 = FB09)

3.3.8 [Y] "Absolute Y Indexed" addressing mode.

This mode is the same as absolute X indexed mode except the Y index register is used. (Note: This mode is available for fewer instructions than "Absolute X Indexed" mode.)

3.3.9 [X)] "Indirect X" addressing mode.

Total instruction length is two bytes. The second byte of this instruction is added to the contents of the X index register to form an eight-bit address (as in the [ZX] addressing mode). The contents of this eight-bit address and the address immediately following are then taken to be the target address.

For example, assume that the second byte of an [X)] instruction is 80 and the contents of the X index register is 3. The contents of memory location 83 contain the lower eight bits of the target address, and the contents of memory location 84 contain the upper eight bits of the target address.

3.3.10 [)Y] "Indirect Y" addressing mode.

Total instruction length is two bytes. The second byte of the instruction contains an eight-bit address. The contents of this address and the following address are taken as a sixteen-bit address (the least significant byte of the sixteen-bit address is in the first byte as in [A]). The target address is formed by adding the contents of the Y index register to the sixteen-bit address.

For example, assume that the second byte of a [)Y] instruction is

80 and the contents of the Y index register is 2. The 6502 first forms a sixteen-bit address by getting its lower eight bits from location 80 (which we will assume contains FF) and its upper eight bits from location 81 (which we will assume contains F0). The contents of the Y index register (3) is then added to the sixteen-bit address (F0FF) resulting in a target address of F102.

It is very important not to confuse [)Y] addressing with [X)] addressing. In practice, [)Y] is used more often and should be understood completely. The [X)] mode is often used with zero in the X index register.

3.3.11 [()] "Absolute Indirect" addressing mode.

Total instruction length is three bytes. The second and third bytes of the instruction contain a sixteen-bit address (the lower eight bits are in the second byte, and the upper eight bits are in the third byte). The location at this address contains the lower eight bits of the target address. The upper eight bits of the target address are in the next memory location. (Note: This mode is only used by the JMP (jump) instruction).

3.3.12 [nn] "Relative" addressing mode.

Total instruction length is two bytes. The second byte of the instruction is treated as a signed two's-complement number (i.e., FF = -1, FE = -2, ..., 80 = -80, 7F = +7F, ...). This signed number is added to the sixteen-bit address of the first byte of the next instruction and the result is the target address. For example, assume that the two bytes of an instruction that uses relative addressing are contained in memory locations F311 and F312. Further, assume that the

second byte of the instruction contains FD. The target address for the instruction is FD + F313 = -3 + F313 = F310. (Note: Relative addressing is used only by branch instructions.)

3.4 Description of 6502 Instruction Set

In this section, we describe all of the instructions recognized by the 6502 microprocessor. For each instruction, the following information is provided:

1. The instruction name--a three letter mnemonic (always capitalized)
2. A description of the operation performed by the instruction.
3. An opcode value for each of the possible addressing modes that can be used with the instruction.
4. The condition codes in the program-status register that are modified by the instruction.

To form a complete instruction, you first use the operation to be performed and the addressing mode desired to select the value of the opcode to be used. The opcode byte is then followed by the (possibly) additional bytes required by the addressing mode being used.

NOTE

> Although there are two hundred and fifty-six possible eight-bit opcodes, many of the possibilities are not used by the 6502. If one of these unused values is accidentally included in a program, unpredictable (often bad) results will occur.

3.4.1 Memory Transfer Instructions

All of the following instructions either copy ("Load") the contents of a memory location into a register or copy ("Store") the contents of a

register into a memory location.

Name	Description
LDA	Load the accumulator with the contents of the specified memory location
STA	Store the accumulator into the specified memory location
LDX	Load the X index register with the contents of the specified memory location
STX	Store the X index register into the specified memory location
LDY	Load the Y index register with the contents of the specified memory location
STY	Store the Y index register into the specified memory location

The opcodes for each memory-transfer instruction with each of its possible addressing modes are as follows:

	[I]	[A]	[Z]	[X)]	[)Y]	[ZX]	[X]	[Y]	[ZY]
LDA	A9	AD	A5	A1	B1	B5	BD	B9	--
STA	--	8D	85	81	91	95	9D	99	--
LDX	A2	AE	A6	--	--	--	--	BE	B6
STX	--	8E	86	--	--	--	--	--	96
LDY	A0	AC	A4	--	--	B4	BC	--	--
STY	--	8C	84	--	--	94	--	--	--

Condition Codes:

Z-flag: set if a value of zero is loaded into a register, cleared otherwise (the Z-flag is not modified by store instructions).
N-flag: set if a negative value is loaded into a register, cleared otherwise (the N-flag is not modified by store instructions).
C and V-flags: not modified by any of these instructions

3.4.2 Register-to-Register Transfer Instructions

These instructions copy the contents of one register into another. Implied addressing is the only addressing mode allowed.

Name	Opcode	Description
TAX	AA	Copy the contents of the accumulator into the X index register.
TXA	8A	Copy the contents of the X index register into the accumulator.
TAY	A8	Copy the contents of the accumulator into the Y index register.
TYA	98	Copy the contents of the Y index register into the

		accumulator.
TSX	BA	Copy the contents of the Stack Pointer into the X index register
TXS	9A	Copy the contents of the X index register into the Stack Pointer.

Condition Codes:

Z-flag: set if the register contents are zero, cleared otherwise (the Z-flag is not modified by the TXS instruction).

N-flag: set if the register contents were negative, cleared otherwise (the N-flag is not modified by the TXS instruction).

C and V-flags: not modified by any of these instructions.

3.4.3 Increment and Decrement Instructions

These instructions add one or subtract one from either a memory location or a register.

Name	Description
INC	Increment the contents of the target address by one.
DEC	Decrement the contents of the target address by one.
INX	Increment the contents of the X index register by one (uses implied addressing mode).
DEX	Decrement the contents of the Y index register by one (uses implied addressing mode).
INY	Increment the contents of the Y index register by one (uses implied addressing mode).
DEY	Decrement the contents of the Y index register by one (uses implied addressing mode).

The opcodes for each increment or decrement instruction with each of its possible addressing modes are as follows:

	[ ]	[A]	[Z]	[ZX]	[X]
INC	--	EE	E6	F6	FE
DEC	--	CE	C6	D6	DE
INX	E8	--	--	--	--
DEX	CA	--	--	--	--
INY	C8	--	--	--	--
DEY	88	--	--	--	--

Condition Codes:

Z-flag: set if the result of the increment or decrement was zero, cleared otherwise.

N-flag: set if the result of the increment or decrement was

negative, cleared otherwise
C and V-flags: not modified by any of these instructions

3.4.4 Add and Subtract Instructions

The 6502 has one addition instruction (ADC) and one subtraction instruction (SBC). The ADC instruction adds the contents of the target to the accumulator and then adds the value of the C-flag to the accumulator. The SBC instruction subtracts the contents of the target address from the accumulator and then subtracts the complement of the C-flag value from the accumulator. The C-flag is added to (or subtracted from) the accumulator to make it easy to perform arithmetic with multiple byte numbers. Whenever arithmetic is done with single byte numbers, the C-flag must be cleared (set) before performing addition (subtraction). After the operation has been performed, the C-flag will be set if an addition resulted in a carry. The C-flag will be cleared if a subtraction resulted in borrow. Another ADC or SBC instruction can then be applied to the next more significant byte of a multi-byte number.

Name	Description
ADC	Add to the accumulator the contents of the specified byte plus the value of the C-flag.
SBC	Subtract from the accumulator the contents of the specified byte. Afterwards, decrement the contents of the accumulator by one if the C-flag was clear.

The opcodes for the addition and subtraction instructions with each possible addressing mode are as follows:

	[I]	[A]	[Z]	[X)]	[)Y]	[ZX]	[X]	[Y]
ADC	69	6D	65	61	71	75	7D	79
SBC	E9	ED	E5	E1	F1	F5	FD	F9

Condition Codes:

Z-flag: set if the final contents of the accumulator were zero, cleared otherwise (not set if decimal mode arithmetic is being used).

N-flag: set if the final contents of the accumulator were negative, cleared otherwise.
C-flag: set if there was a carry in addition, cleared if not; cleared if there was a borrow in subtraction, set if not.
V-flag: set if arithmetic overflow occurred, cleared if not.

3.4.5 Shift and Rotate Instructions

These instructions move each bit in the selected byte (either a memory location or the accumulator) one place to the left or right. The vacated bit position (bit 7 for a right shift, bit zero for a left shift) is filled either with a zero or with the previous value of the C-flag. In all cases, the bit that is bumped off the end is placed in the C-flag.

Name	Description
ASL	Arithmetic shift left. Shift all the bits in the targeted byte left one place, shift bit 7 into the C-flag, and shift a zero into bit zero. (This operation is equivalent to multiplying by two).
LSR	Logical shift right. Shift all bits in the targeted byte right one place, shift a zero into bit 7, and shift bit zero into the C-flag.
ROL	Rotate left. Shift all bits in the targeted byte one place to the left, shift the C-flag into bit zero, and shift bit 7 into the C-flag.
ROR	Rotate right. Shift all bits in the targeted byte one place to the right, shift the C-flag into bit 7, and shift bit zero into the C-flag. (Note: if the C-flag is set to the same value as bit 7 before this instruction is executed, the result of the ROR will be to divide either a positive or negative number by two.)

The opcodes for the shift and rotate instructions with each of the possible addressing modes are as follows:

	[A]	[Z]	[ZX]	[X]	Accumulator
ASL	0E	06	16	1E	0A
LSR	4E	46	56	5E	4A
ROL	2E	26	36	3E	2A
ROR	6E	66	76	7E	6A

Condition Codes:

Z-flag: set if the result of the operation was zero, cleared otherwise.

N-flag: set if the result of the operation was a negative value, cleared otherwise.
C-flag: set to the value of the bit that was "shifted out" of the byte being shifted.
V-flag: not modified by any of these instructions.

3.4.6 Logical Operation Instructions

These instructions perform a bit-by-bit logical operation between the bits of a specified byte and the bits of the accumulator, and leaves the result in the accumulator. A bit-by-bit operation means that each bit of the final accumulator value is a logical function of the same bit in the initial accumulator and in the specified byte. The instructions available and the logical functions they provide are listed below.

Name	Description
AND	Perform bit-by-bit "AND" of the accumulator with the selected byte and place the result in the accumulator.

```
1 "AND" 1 = 1
1 "AND" 0 = 0
0 "AND" 1 = 0
0 "AND" 0 = 0
```

For example, A3 "AND" 8A = 82

ORA — Perform bit-by-bit "OR" of the accumulator with the selected byte and place the result in the accumulator

```
1 "OR" 1 = 1
1 "OR" 0 = 1
0 "OR" 1 = 1
0 "OR" 0 = 0
```

For example, A3 "OR" 8A = AB

EOR — Perform bit-by-bit "EOR" ("exclusive OR") of the accumulator with the selected byte and place the result in the accumulator.

```
1 "EOR" 1 = 0
1 "EOR" 0 = 1
0 "EOR" 1 = 1
0 "EOR" 0 = 0
```

For example, A3 "EOR" 8A = 29. (Note that "NOT" (that is, the logical operation that changes all zeros to ones and all ones to zeros) can be performed by performing an "EOR" operation on the accumulator with a value of FF.)

The opcodes for each logical instruction with each possible addressing mode are as follows:

	[I]	[A]	[Z]	[X)]	[)Y]	[ZX]	[X]	[Y]
AND	29	2D	25	21	31	35	3D	39
ORA	09	0D	05	01	11	15	1D	19
EOR	49	4D	45	41	51	55	5D	59

Condition Codes:

Z-flag: set if the final contents of the accumulator are zero, cleared otherwise.
N-flag: set if the final contents of the accumulator are negative, cleared otherwise.
C and V-flags: not modified by any of these instructions.

3.4.7 Stack Instructions

The basic stack operations performed by the 6502 are "push" and "pull". Push puts a new byte onto the stack, and "pull" removes the last byte pushed onto the stack. The stack pointer register always points to the address where the next byte will be pushed.

The stack is used automatically when calling (JSR instruction) or returning from (RTS instruction) a subroutine. Therefore any bytes pushed onto the stack within a subroutine must be pulled from the stack before the return instruction is executed. This is referred to as "keeping the stack balanced".

Name	Opcode	Description
PHA	48	Push the contents of the accumulator onto the stack.
PLA	68	Pull the next byte from the stack and put it into the accumulator.
PHP	08	Push the contents of the processor status register onto the satck.
PLP	28	Pull the next byte from the stack and place it into the processor status register.

Condition Codes:

PHA and PHP do not change any condition codes
PLA changes the Z-flag and N-flag to indicate if the byte pulled from the stack was zero or negative.
PLP loads a byte into the processor status register, and thus, sets a value into each of the condition codes (as well

as setting the values for all the other flags). See Section 3.2.4 for the meaning of each bit in the processor status register.

Besides the above instructions, the only other instructions that affect the stack are JSR, RTS, BRK, and TXS.

3.4.8 Compare Instructions

These instructions are used to compare the contents of the accumulator with the value of another byte. After the compare operation has been performed, neither the contents of the register nor the value of the byte being compared will have been changed--the only thing changed will be the values of the Z, N, and C-flags. These flags will be set as if the contents of the byte being compared had been subtracted from the register.

The purpose of the compare instructions is to set up the condition codes for subsequent use with a conditional branch instruction. The three 6502 compare instructions are as follows:

Name	Description
CMP	Compare the contents of the accumulator with the specified byte and set the condition codes accordingly.
CPX	Compare the contents of the X index register with the specified byte and set the condition codes accordingly.
CPY	Compare the contents of the Y index register with the specified byte and set the condition codes accordingly.

The opcodes for each compare instruction with each of its possible addressing modes are as follows:

	[I]	[A]	[Z]	[X)]	[)Y]	[ZX]	[X]	[Y]
CMP	C9	CD	C5	C1	D1	D5	DD	D9
CPX	E0	EC	E4	--	--	--	--	--
CPY	C0	CC	C4	--	--	--	--	--

Condition Codes:

Same as SBC instruction except the V-flag is not altered.

3.4.9 Branch Instructions

The 6502 instruction set includes eight conditional branch instructions. There are two instructions for each of the four condition code flags-- one instruction to test if the flag was set, and one instruction to test if the flag was clear. If the test made is true, the program counter is reset to the address specified by the second byte of the instruction (using the "relative" addressing mode discussed in Section 3.3.12). If the test is false, the program counter is not changed and the instruction after the branch instruction is executed next.

The conditional branch instructions are listed below:

Name	Opcode	Description
BMI	30	Branch if minus (i.e., branch if N-flag is set).
BPL	10	Branch if plus (i.e., branch if N-flag is clear).
BNE	D0	Branch if not equal to zero (i.e., branch if Z-flag is clear).
BEQ	F0	Branch if equal to zero (i.e., branch if Z-flag is set).
BCC	90	Branch if C-flag is clear.
BCS	B0	Branch if C-flag is set.
BVC	50	Branch if V-flag is clear.
BVS	70	Branch if V-flag is set.

Condition codes:

Not altered by any of these instructions.

3.4.10 Jump Instruction

This instruction forces the 6502 to reset the program counter to a new value and, thus, to continue program execution at a different address. This is the only instruction to use the "indirect" addressing mode. The opcodes for the jump instruction are as follows:

	[A]	[()]
JMP	4C	6C

Condition Codes:

None of the condition codes are changed.

3.4.11 Subroutine Call and Return Instructions

There are two 6502 instructions used with subroutines--one subroutine "call" instruction (JSR) and one subroutine return instruction (RTS). A subroutine is a series of instructions that can be "called" from one or more places within a program. After the instructions in the subroutine have been executed, a "return" is made (i.e., the microprocessor continues program execution at the address following the subroutine call instruction).

The subroutine calling instruction pushes the value of the program counter minus one onto the stack (the least significant byte is pushed first). This keeps a record on the stack of from what address the subroutine was called. When a return is made from a subroutine, a two byte address is removed from the stack, the address is incremented by one, and a jump made to the resulting address. Program execution then proceeds starting with the first instruction after the call to the subroutine.

One subroutine can call other subroutines (even itself !) as long as the stack does not overflow its allotted memory region. The stack can be used during a subroutine but it must be restored to the entry configuration before the RTS instruction is executed.

In summary, the subroutine instructions are:

Name	Opcode	Description
JSR	20	Jump to subroutine. Push the current value of the program counter minus one onto the stack (least significant byte first), then jump to the address of the subroutine. This instruction uses absolute addressing mode to specify the address of the subroutine.
RTS	60	Return from subroutine. Pull a two-byte absolute address from the stack, increment the address by one, and jump to the address. This instruction is one byte long and uses implied addressing.

Condition Codes:

None of the condition codes are changed by these instructions.

3.4.12 Status Flag Manipulation Instructions

These instructions set and clear flags in the processor status register. All of these instructions use the implied addressing mode.

Name	Opcode	Description
CLC	18	Clear C-flag
SEC	38	Set C-flag
CLD	D8	Clear D-flag
SED	F8	Set D-flag
CLI	58	Clear I-flag
SEI	78	Set I-flag
CLV	B8	Clear V-flag

Condition Codes:

None of these instructions alter any of the conditions except for those flags mentioned in the instruction's description.

Note: The V-flag can be set with the BIT instruction.

3.4.13 NOP instruction

NOP is a single byte instruction (with opcode EA) that does nothing. It is useful for waiting for a while (a few microseconds) or to reserve space in a program for instructions that you may want to insert later. Similarly, if an instruction needs to be removed, you can replace each of its bytes with NOP instructions to avoid having to move other instructions to close the gap.

3.4.14 Bit Test Instruction

The bit test instruction (BIT) performs a bit-by-bit "AND" of the accumulator with a specified byte (See description of the AND instruction for a description of a "bit-by-bit AND".). The results of

this AND is only used to set the condition codes, the contents of the accumulator as well as the value of the specified byte are not changed. Like the compare instructions, the BIT instruction is usually followed by a conditional branch instruction. The opcodes for the BIT instruction with each of its possible addressing modes are as follows:

	[A]	[Z]
BIT	2C	24

Condition Codes:

Z-flag: set if the result of the bit-by-bit AND is zero, cleared otherwise.
N-flag: set to the value of bit 7 of the byte being ANDed with the accumulator.
V-flag: set to the value of bit 6 of the byte being ANDed with the accumulator.
C-flag: not changed by this instruction.

3.4.15 Break Instruction

The break instruction (BRK) is the only part of the interrupt capability of the 6502 available to MagiCard users. Execution of the BRK instruction (opcode 00) performs a reset of the MagiCard similar to that caused by turning the power off and on again. BRK has the following advantages over turning the power off:

1. A reset can be initiated under program control.
2. The contents of the program storage memory (F000 - F3FF) are not lost.

A programming mistake will often lead to the execution of a BRK. For example, if a jump is made to an improper location, the 6502 will start to execute some random values as if it were a program. In such cases it is reasonably likely that it will soon encounter a byte containing a zero, and therefore, reset itself via a BRK. A complete description of the reset that occurs when a BRK is executed (or when the MagiCard is first powered up) will be found in Section 7.16.1.

CHAPTER 4

MagiCard Keyboard Functions

In this chapter we discuss in detail the function of each keyboard controller key when the MagiCard is plugged into your Atari game. The keys on the keyboard controllers are referred to by a letter "L" or "R" ("L" for the left controller, "R" for the right controller) followed by a symbol for the key on the controller (0,1,2,...,9,*,#). The function names correspond to the names on the cut-out keyboard overlays.

Each key on the left controller has two distinct functions (referred to as "unshifted" and "shifted"). Pressing the R0 key (right controller zero) key causes all the keys on the left controller to take on their shifted functions. Shifted mode is exitted by pressing the L0 key. The right controller keys have no function in shift mode. The shifted functions for each left controller key are printed on the keyboard overlay above the unshifted function.

The first section in the chapter gives a brief description of the function of each controller key. Each subsequent section is devoted to describing one particular key function in more detail.

4.1 Brief Description of Key Functions

4.1.1 Left Controller Functions (Unshifted)

Key	Name	Function
L1	1	Enter hex digit "1" into the right most character of the display's top line.
L2	2	Enter hex digit "2" into the right most character of the display's top line.
L3	3	Enter hex digit "3" into the right most character of the display's top line.
L4	4	Enter hex digit "4" into the right most character of the display's top line.
L5	5	Enter hex digit "5" into the right most character of the display's top line.
L6	6	Enter hex digit "6" into the right most character of the display's top line.
L7	7	Enter hex digit "7" into the right most character of the display's top line.
L8	8	Enter hex digit "8" into the right most character of the display's top line.
L9	9	Enter hex digit "9" into the right most character of the display's top line.
L0	0	Enter hex digit "0" into the right most character of the display's top line.
L*	Fetch	Read the contents of the memory location whose address is in the "Fetch Address" (see description of R* key) and place into the right two hex digits of the display's top line, shift the previous right two display digits two digits to the left, and increment the "Fetch Address" by one.
L#	Store	Write the contents of the right two digits on the display's top line into the memory location whose address is in the "Store Address" (see description of R# key) and increment the "Store Address" by one.

4.1.2 Right Controller Functions (Unshifted)

Key	Name	Function
R1	A	Enter hex digit "A" into the right most character of the display's top line.
R2	B	Enter hex digit "B" into the right most character of the display's top line.
R3	C	Enter hex digit "C" into the right most character of the display's top line.
R4	D	Enter hex digit "D" into the right most character of the display's top line.
R5	E	Enter hex digit "E" into the right most character of the display's top line.
R6	F	Enter hex digit "F" into the right most character of the display's top line.
R7	Cend Ad	Enter the right four hex digits on the display's top line into "Cend Address". "Cend Address" is used when writing data to or reading data from a cassette.
R8	Relc	Calculate relative offset for branch instructions. As discussed in Section 3.3.9, the second byte of each branch instruction contains a signed two's-complement number that specifies the number of bytes between the address following the branch instruction and the address to be branched to. This byte is called the "relative offset" of the branch. You can use the MagiCard to determine the value to use for the relative offset by proceeding as follows. First, place the absolute address (four hex digits) of the second byte of the branch instruction into "Store Address". Next, enter the absolute address of the destination of the branch into the right four digits on the display's top line. Pushing the R8 key then calculates the two hex digit branch offset that should be stored in the second byte of the branch instruction. A result of FF implies that the branch is out of range. The previous contents of the right four digits on the display's top line are moved into the left four digits.
R9	Extra Ad	Enter the right four hex digits on the the display's top line into "Extra Address". The current value of "Store Address" is also placed into the left four digits of the display's top line. "Extra Address" is not used by the MagiCard Monitor; however, it can be useful to pass information to a user program.
R0	Shift	Shift lock. Pressing this key causes each key on the left controller to assume its "shifted" function. The left controler remains "shifted" until the L0 ("unshift") key is pressed. All keys on the right controller have no "shifted" function. Pressing the "Game Reset" on the

Video Computer console will also "unshift" the keys.

R* Fetch Ad Enter the right four hex digits on the display's top line into "Fetch Address". The current value of "Cend Address" is also placed into the left four digits of the display's top line.

R0 Store Ad Enter the right four hex digits on the display's top line into "Store Address". The current value of "Extra Address" is also placed into the left four digits of the display's top line.

4.1.3 Left Controller Functions (Shifted)

These functions are performed by the left controller keys when it has been "shifted" by pressing the R0 key. The functions performed by keys L1 through L6 are specially protected to avoid accidental activation. When one of these keys is pressed, the TV screen will go blank. If at this point the "Game Select" switch on the Video Computer console is pressed, the selected function will be executed. If, on the other hand, the "Game Reset" switch on the Video Computer console is pressed, the MagiCard will <u>not</u> perform the selected function but will instead perform its power-up initialization sequence.

Key	Name	Function
L1	Run	Start program execution at the address contained in "Store Address". (Waits for "Game Select" with screen blank.)
L2		No operation (Waits for "Game Select" with screen blank.)
L3		No operation (Waits for "Game Select" with screen blank.)
L4	Cwrite	Cassette write. Write to cassette the contents of memory addresses "Fetch Address" to "Cend Address" -1. See Section 4.6 for details about cassette operations. (Waits for "Game Select" with screen blank.)
L5		No operation (Waits for "Game Select" with screen blank.)
L6	Cread	Cassette read. Read data from cassette and store into memory addresses "Fetch Address" through "Cend Address" -1. See section 4.7 for details about cassette operations. (Waits for "Game Select" with screen blank.)
L7	Hex Dump	Hexadecimal dump. Display on the screen (as four hex digits) the contents of two bytes of memory starting at the address contained in "Fetch Address". "Fetch Address" is then incremented by two.
L8	Ins Dump	Instruction Dump. Display on the screen the disassembled form of the instruction starting at the address contained in "Fetch Address". "Fetch Address" is then incremented by the length of the instruction.
L9		No operation
L0	Unshift	"Unshift" the left controller keys. The left controller keys return to their normal "unshifted" functions.
L*		No operation
L#		No operation

4.2 Hex Dump Function (Shift L7) -- Further Information

After pressing the R0 (Shift) key, each time the L7 (Hex Dump) key is pressed the entire display is moved up one line (scrolled one line) and a new line is added at the bottom. The new line contains three pieces of information (or "fields"). The first field is four characters in length and displays the "Fetch Address" value as a hex number. The second field is two characters in length and displays the hex number contained in the location at that address. The third field is also two characters in length and displays the hex number contained in the next location. After the new line is displayed, the contents of "Fetch Address" is incremented by two so the next time the L7 key is pressed the contents of the next two memory locations will be displayed.

4.3 Instruction Dump Function (Shift L8) -- Further Information

After pressing the R0 (Shift) key, each time the L8 key is pressed, the entire display is scrolled two lines up and two new lines are added at the bottom. These two new lines describe the instruction contained at address "Fetch Address". The upper new line contains the four digit hex address of the instruction being dumped. The next line contains three fields--the instruction name field, the instruction addressing mode field, and the address information field.

The instruction name field occupies the first three characters. This field contains the three character opcode name (see Section 3.4) for the instruction being dumped (i.e., the instruction starting at address "Fetch Address"). If the byte at the address "Fetch Address" is not a legal opcode, the instruction name "ILL" is displayed and the rest of the line is blank.

The addressing mode field occupies the next two characters. This field contains the two character addressing mode name for the

instruction (see Section 3.3). For relative addressing, this field will contain the two hex digits of the instruction (which is the relative address itself).

NOTE

> Due to an irregularity in the 6502 instruction set, the addressing mode of the "JSR" instruction will always be listed as [Y] rather than the correct value of [A]. Just ignore this error and remember that JSR (opcode 20) has an addressing mode of [A].

The additional addressing information field occupies the last four characters. For instructions with accumulator or implied addressing (i.e., an addressing mode field of [ ]), this field will be blank. For two byte instructions (with the exception of branch instructions), this field will contain the second byte of the instruction. For three byte instructions, this field will contain the four hex digit address contained in the last two bytes of the instruction. The most significant byte of the address is placed in the first two hex digits (for easy reading); even though they are stored in the opposite order in memory. For branch instructions, this field will contain the absolute address to which the instruction would branch. Hence, you can see both the relative address (in the addressing mode field) and the corresponding absolute address (in the additional address information field).

After the instruction information is displayed on the bottom two lines, "Fetch Address" is incremented by the length of the instruction (Instructions with illegal opcodes are considered to have a length of one byte). Note that attempting to dump arbitrary data as instructions may be misleading because any random one byte number has about a 60% chance of being a legal opcode. Similarly, if you begin

to dump at an address that corresponds to the second or third byte of a multiple byte instruction, that byte may be interpreted as a legal instruction opcode. Depending on the particular case, several spurious instructions may be dumped (some of them "ILL") before, by chance, the first byte of a real instruction is encountered. From that point on, dumping will proceed normally.

The same sort of trouble may occur if you are dumping a program that you have incorrectly entered into memory. A mistyped opcode to a multiple byte instruction may give rise to a confused dump immediately following the error. In this case, you should correct the erroneous opcode and dump again. These cases may sound confusing, but once you are aware of the possibility of their occurrence, they are easy to recognize in practice.

4.4 Combined Use of Fetch (L*) and Store (L#) Keys

The "Fetch" key places the byte whose address is "Fetch Address" into the right two hex digits of the display's top line. Similarly, the "Store" key stores the right two hex digits of the display's top line into the address "Store Address". By setting "Fetch Address" and "Store Address" to the same value, and then alternately pushing "Fetch" and "Store", you can move through memory while watching the bytes move through the right two characters on the display's top line.

It is also possible to make changes to memory as the bytes are displayed. After a byte has been placed into the right two hex digits of the display by pressing the "Fetch" key, a new value may be keyed into the display (by pressing the keys L0 through L9 or R1 through R6) before the "Store" key is pressed. When "Store" is pressed, the old value for the byte will be overwritten by the new value.

If the "Fetch Address" and "Store Address" are set to different values, an identical procedure will move data from one place in memory

to another (with optional modification to any byte desired). It is helpful to understand that the above procedures are simply combinations of the simple functions of "Fetch" and "Store".

4.5 Relc Function (R8) -- Further Information

It is often convenient to write a program using names for the addresses that are the destinations of branch, JMP, or JSR instructions. Once the program has been written, the value of each address name can be determined by choosing a starting address for the program and counting the number of bytes in all the instructions between the starting address and each address name. For any instruction that uses absolute addressing (e.g., JMP or JSR), the absolute address itself is placed in the second and third bytes of the instruction. In the case of branch instructions, however, the absolute address itself is not included in the instruction. What is needed is a two's complement byte offset between the address of the memory location immediately after the branch instruction and the address of the branch destination. For branches to nearby locations, the offset calculation becomes easier with practice. Branches to distant parts of the program are not so easily handled.

The "Relc" function can save time in determining the offset to use for branch instructions (read its description in Section 4.1.2 if you have not already done so). When you are entering a branch instruction as part of a program, first key in and store in memory the opcode byte for the branch. Then, instead of the two digit byte offset that goes into the second byte of the branch instruction, key in the absolute address to which the branch is intended to go and press the R8 ("Relc") key. If the new value in the right two characters of the display is not FF, it is the correct relative offset for the branch instruction. You only need to press the "Store" key to

place the offset into memory and can then continue keying in the next instruction of the program.

4.6 Cassette Write Function (Shift L4) -- Further Information

The "shifted" function of the L4 key writes a range of memory locations to a cassette tape. The series of operations required are as follows:

1. Enter the address of the first byte to be saved on cassette into "Fetch Address" (using the R* key).
2. Enter the address immediately following the last byte to be saved into "Cend Address" (using the R7 key).
3. Press "Shift Lock" (R0), then "Cwrite" (L4). (The display will go blank.)
4. Flip the cassette interface switch on.
5. Start the cassette recorder in record mode (Volume half way up, tone on "high").
6. Press the "Game Select" switch on the Video Computer console (If you are writing at the beginning of the cassette, wait until the leader is passed before pressing "Game Select").
7. When the display reappears, wait a few seconds and stop the recorder.
8. Flip the cassette interface switch off.
9. Press the "Unshift" key (L0).

Writing data to a cassette produces a 12 second high-pitched header tone followed by a high warble tone that represents the data. The data tone is followed by a loud, low buzz tone. Each byte written to cassette has an associated parity bit (even parity) which is checked when the byte is read back ("Cread" function). Details of the setup and connection of the cassette are included along with construction notes in Appendix E.

4.7 Cassette Read Function (Shift L6) -- Further Information

The "shifted" function of the L6 key reads data from a cassette tape

into a range of memory locations. The series of operations required are as follows:

1. Enter the address of the first memory location to receive data from cassette into "Fetch Address" (using the R* key).
2. Enter the address of the memory location immediately following the area to receive data into "Cend Address" (using the R7 key).
3. Press the "Shift" key (R0), then the "Cread" key (L6). (The display will go blank.)
4. Flip the cassette interface switch on.
5. Start the cassette in playback mode (Volume halfway up, tone "high") with the tape positioned within the high-pitched "header".
6. Press the "Game Select" switch on the Video Computer console (while the recorder is still playing back the header tone).
7. When the display reappears, stop the recorder.
8. Flip the cassette interface switch off.
9. Press the "Unshift" key (L0).
10. Press the L0 key four times and then press the "Cend Ad" key (R7).

After all these steps are done, the left four characters on the display's top line will show the address of the location immediately after the last memory location that was filled with cassette data. This should equal the value placed into "Cend Address" in Step 2. If it does not, bad data was encountered (a parity error) at about the address shown and the cassette input stopped. Try rereading, perhaps with the volume turned up a bit (the tone control should be as high as possible).

If the display doesn't return after the recorder has read through the data, either the recorder volume was too low, a cable was disconnected, or an improper address was entered in Step 1 or Step 2. The simplest way to recover is to briefly turn off the power to the Video Computer System. If that is not desirable, playing the cassette

through any random data while adjusting the volume will usually cause recovery of the display (due to an input parity error).

Notice that data can be read back into memory at any address entered in Step 1--not necessarily the same place it resided when it was recorded. Similarly, less than the full amount of data can be read back from a tape (though you must always start at the beginning). For example, a gap can be created in the middle of a program to add new instructions. To produce the gap, first read the entire program into memory but store it at an address that is larger than the original starting address by the size of the desired gap. Next, read the program in again storing it at the normal starting address but ending at the instruction just before the gap. This leaves a region of duplicated instructions in the middle of the program that can be overwritten by keying in new instructions. It is also possible to delete instructions from the middle of a program by reading the entire program into a lower address than usual and then overwriting the beginning by reading in a portion of the program at the normal address.

Code moved in this way may require a few modifications (i.e., to addresses contained within its instructions or to another portion of the program that references the moved portion), but for a large program of hundreds of instructions this can be far easier than reentering the entire program. You may even develop a few subroutines that are used frequently in different programs. If you write them using a carefully chosen subset of the 6502 instruction set and addressing modes, you will be able to place them anywhere in memory (using the cassette if they are long) and use them without modifications. A subroutine having this property is said to be "position independent".

CHAPTER 5

MagiCard Memory Map

In this chapter we define the MagiCard memory map--that is, what each possible two-byte absolute address is used for. As you will see, some addresses are used to refer to memory locations (such as the memory that your programs can be stored in) while other addresses are used to control the audio and video capabilities of your Atari Video Computer System. In the next section, we give a broad outline of the use of each range of addresses. In the remaining sections of this chapter we provide more details of the use of those addresses that reference actual memory. A description of the audio and video control features is given in Chapter 6.

5.1 Basic Memory Map

Address Range	Contents
0000 - 003F	Control registers for the TV display, sound and part of the game controllers. This range of addresses controls all of the video and audio features of your Video Computer System; therefore it is called the "Display Generator".
0040 - 007F	The same as 0000 - 003F (i.e. a reference to location 0040 is identical to a reference to location 0000; a reference to 0041 is identical to a reference to 0001 etc.).
0080 - 00FF	Read/Write memory (RAM) used for the stack, temporary storage of calculations, etc. This range of memory locations is called "Zero Page RAM". The MagiCard Monitor uses locations 0080 - 00A9 (as temporary variables) and 00F9 - 00FF (for the stack) so use of these locations by a user program should be avoided.
0100 - 01FF	Same as 0000 - 00FF
0200 - 02FF	Control registers for the 6532 multifunction device. This device is described in Section 6.1.
0300 - EFFF	This address range serves no useful function but no reference should be made to these addresses as it may cause unpredictable results for your program !!
F000 - F3FF	Program storage read/write memory (RAM). Locations F000 - F0D1 are used to store the pixel-bits used by the TV display generating subroutine (see Section 7.1). User written programs can occupy locations F0D2 - F3FF. This entire memory range (F000 - F3FF) can be freely read by a user program but any attempt to modify one of these locations must occur via one of the monitor subroutines (see Section 5.2).
F400 - F7FF	Used by the MagiCard Monitor program to store values into the RAM contained in F000 - F3FF (see Section 5.2).
F800 - FFFF	Contains read only memory (ROM) which holds the MagiCard Monitor program and its user callable subroutines. This memory can be read by the user program but cannot be written into.

5.2 Details on Use of Address F000 - F7FF

As mentioned above, the address range F000 - F3FF contains memory that can be used for user written programs and user data. However there are important restrictions on how this memory can be used. The restrictions are as follows:

1. No instructions that store into memory can reference an address in the range F000 - F300 (These addresses are "read only"). An example of a store instruction is "STA".

2. An instruction that stores a value into an address in the range F400 - F7FF will actually place the value into an address 400 lower in memory. For example, a store to location F523 places the value into F123. (The addresses F400 - F7FF are "write only").

3. None of the following instructions can be used with any addresses in the range F000 - F7FF : ASL, DEC, INC, LSR, ROL, ROR. The reason for this is that these instructions read a value from a location, modify it, and store it back to the same location. Since addresses from F000 - F7FF contains either read only or write only memory, it is impossible to read from and then write back into memory using the same address for both operations.

4. Instructions that store to the address range F400 - F7FF must be located in the MagiCard Monitor ROM or the zero page RAM (80 - FF). Any other attempt to store into these locations may work with some MagiCard modules but its reliability cannot be guaranteed. User programs that wish to store into locations F400 - F7FF can either do so from the zero page RAM (perhaps by placing a small subroutine there) or they can make use of the monitor subroutines that store into this address range (in particular, see the STPM subroutine described in Section 7.2).
   Additional Restriction: When using indexed addressing modes to store into this address range, the most significant byte of the address provided by the instruction must not change when the index is added.

5.3 Use of Zero Page RAM (0080 - 00FF) by MagiCard Monitor

In this section we give a brief description of how the zero page RAM is used by the MagiCard monitor program. The four letter names for subroutines (such as PSHD and DSPL) used below are the same ones used in Chapter 7 where the monitor subroutines are described. The symbols "LSB" and "MSB" mean "least significant byte" and "most significant byte" respectively.

Address	Use
80,81	LSB,MSB of keyed in digits accumulated by subroutine PSHD.
82,83	Left,Right keyboard controller number of new key pressed (filled by subroutine DSPL).
84,85	LSB,MSB of "Store Address" (filled by the monitor when the R0 key is pressed).
86,87	LSB,MSB of "Fetch Address" (filled by the monitor when the R* key is pressed).
88,89	LSB,MSB of "Extra Address" (filled by the monitor when the R9 key is pressed).
8A,8B	Left,Right keyboard controller number of current key pressed (filled by subroutine DSPL).
8C,8D	LSB,MSB of "Cend Address" (filled by the monitor when the R7 key is pressed). Also used as the base address for subroutine STPM.
8E	Character number used as input for subroutine ONEC. Also horizontal position input parameter for subroutine CALP.
8F	Line number used as input for subroutines ONEC and DOLN. Also vertical position input parameter for subroutine CALP.
90	Used by CRED/CWRT to hold the byte currently being read from/written to cassette. Used as a temporary storage location by ONEC, DOLN, CALP and DISA.
91	Used by CRED to hold the LSB of the address where the next byte read from the cassette is to be stored. Used as a temporary storage location by ONEC, DOLN, and MAIN.
92	Used by CRED to hold the MSB of the address where the next byte read from the cassette is to be stored. Used as a temporary storage location by ONEC, DOLN, and MAIN.
93 - 97	Used by subroutines ONEC and DOLN.
93	Used by ADTY and INSN to save the X index register. Used by CRED and CWRT in performing parity checks.
94,95	Used by GO (see Section 7.15.7) to store the LSM,MSB of the address to which it jumps.
96	Cursor blink counter used by MAIN.
98 - A1	ASCII character codes input to ONEC and DOLN.
A2 - A9	Zeroed by RSET (when a BRK instruction is executed or when the Video Computer System is powered up) but not otherwise used by the monitor.
A9 - F8	Totally untouched and unused by the MagiCard monitor. Can be used by a user program for anything it wishes.

F9 - FF Used as the stack by the monitor and monitor subroutines. When execution of a user program is begun (by pushing the "Run" key), the stack pointer register will contain FF. Monitor subroutines are guaranteed not to push more than seven bytes onto the stack, thus using locations FF through F9. If the user program uses the stack and then calls a monitor subroutine, the stack may extend below location F9. Be very cautious to avoid placing a program in locations that may get overwritten by the stack (or vice-versa).

CHAPTER 6

Details of Video Computer System Features

The Atari Video Computer System is capable of performing many sophisticated tasks as you can immediately see by playing an Atari game cartridge. The MagiCard monitor provides subroutines that allow easy access to a very useful subset of the Atari features. In order to fully realize the potential capabilities, however, a much deeper understanding of how the Video Computer System works is required.

In this chapter a great deal of information about the operation of the Video Computer System is given. For some features (such as the "high resolution" graphics capability), a complete description is beyond the scope of this manual. However, you will find enough of an explanation to give you a good start in investigating these features yourself (either by using the MagiCard monitor keyboard to store values into the control registers, or by writing some test programs).

The first three sections of this chapter each discuss one of the devices that are used to provide the Video Computer System features. These devices are the 6532 multifunction integrated circuit, the game controllers (keyboard, paddle, and joystick) and switches, and the Atari display generator integrated circuit. The last two sections give more details on the sound generating feature and on how to keep the display synchronized on your TV set.

6.1 The 6532 Multifunction Integrated Circuit

The 6532 multifunction integrated circuit (called the "RIOT") is a chip in the 6502 family of integrated circuits. It is manufactured by the same companies that make the 6502 (Rockwell International and MOS Technology). The name "RIOT" stands for RAM-Input-Output-Timer, because the RIOT does all these things. Related chips in the 6502 family (such as the 6522 PIA and especially the 6530 RRIOT) are similar enough to the RIOT that you can learn a great deal by reading any information you may be able to find about them. In this section the RIOT is described, slighting its interrupt capabilities, because interrupts are not used by the Video Computer System. The details of how the RIOT is connected to the Atari controllers is given in Section 6.2.

6.1.1 Input/Output (I/O) Ports

The RIOT has two input/output ports, labeled "A" and "B". Each I/O port has eight lines that can either input (read) or output (write) TTL logic signals (i.e. from 0 to 5 volts). Each port also has two eight-bit registers: a "data direction register" and a "data value register". The n'th bit in these registers is associated with the n'th line in the port. The data direction register is used to specify the mode (input or output) for each line. A bit value of one sets the corresponding line to output, a value of zero sets it to input. Once the data direction register has been set, the data itself is read from (or written to) the data value register. When using a port for output, a bit set to one in the data value register produces an output signal of TTL "high" (2.4 to 5.0 volts). A bit set to zero produces an output signal of TTL "low" (0.0 to 0.4 volts). Each output line can drive one TTL load plus 30 pf of capacitance. When using a port

for input, a TTL high signal or an unconnected line produces an input bit of one, and a TTL low signal produces an input bit of zero.

On power up, the RIOT data direction registers are set to zero (input). All eight bits of the A port are set to output when the DSPL subroutine is called (and by the monitor program itself because it calls DSPL). The B port is left in the input mode by the monitor program and is also reset to input when a BRK instruction is executed.

The A and B I/O ports are very similar in their operation, with the following execeptions:

1. When used for output, I/O port B can supply more current at logic state zero than port A (Both ports can supply 1.6 ma at 0.4 volts. Port B can supply 3.0 ma at 1.5 volts).

2. If a line in the B port is set to output, the value of its bit in the data value register will always be the last value the bit was set to. A read of a bit in the A port data value register will, under the same circumstances, return the logic value that actually exists on the output line. Thus, if the A port's smaller drive capacity is overridden by an incoming signal, the incoming signal can be detected. Reading the data direction register of either port always returns the last value written into it (00 immediately after power up).

3. The most significant bit (bit seven) of the A port has the capability of detecting logic transitions and seting another bit in a RIOT control register when a transition has occurred. It is possible to detect transitions from logic zero to logic one, logic one to logic zero, both, or neither.

6.1.2 RAM

The RAM contained in the RIOT is just the zero page RAM with memory addresses 80 through FF.

6.1.3 Timer

The RIOT has a single count-down timer. A somewhat simplified description of its operation is given. (The timer in your Video

Computer System may not do exactly what the 6532 data sheet says, but it will do what is described here.) Basically the timer is a register whose value can be set, and which will then automatically decrement at some fraction of the microprocessor clock rate. By using RIOT control registers you can:

1. Set the timer to a value and begin its countdown.
2. Set the rate of countdown as a fraction of the microprocessor clock rate (1.197MHz).
3. Read the current timer value.
4. Check if the countdown has reached zero.

When the countdown reaches zero, the timer sets the timer-count-down flag, waits as long as it did at any other timer value during the countdown, sets the timer value to FF, and begins a repeated count down from FF to 00 at the full microprocessor clock rate.

NOTE

> The microprocessor clock counts 76 times in 1/15,750 of a second. 1/15,750 of a second is the time needed for a TV set to scan one horizontal line of the picture. This fact is important when you are trying to synchronize a display on the TV screen (see section 6.5).

6.1.4 RIOT Control Registers

In this section the addresses and functions of the RIOT control registers are described. These registers are memory locations that perform a special function when they are written into or read from. Many of the registers serve a dual function--when they are written into they may set a value for one of the RIOT functions, and when they are read from they may give a value that is associated with a

different RIOT function. The RIOT cannot interrupt the 6502 microprocessor, but the the interrupt-status flags (in location 0285) will be set whenever an interrupt-causing condition has occurred. When using the table below, remember that the bits in a byte are numbered from zero (the least significant bit) through seven (the most significant or sign) bit.

Address	(Read/Write)	Function
0280	(R/W)	I/O port A data value register. When a line that has been set to output is read, the value seen is the actual logic value on the line not the value that was last written into this register for that line.
0281	(R/W)	I/O port A data direction register. (A bit value of zero sets a line for input.)
0282	(R/W)	I/O port B data value register. When a line that has been set to output is read, the value seen is the value last written into this register for that line.
0283	(R/W)	I/O port B data direction register. (A bit value of zero sets a line for input).
0284	(R)	Current timer contents. This value is always being counted down.
0294	(R)	Same as 0284 except reading this location disables setting of the timer-count-down flag when the timer has counted down to zero.
029C	(R)	Same as 0284 except reading this location enables setting of the timer-count-down flag when the timer has counted down to zero.
0285	(R)	Read and reset flag bits Bit seven is the timer-count-down flag (set to one when the timer reaches zero). Bit six is the bit-seven-transition flag. This bit can be used (assuming that things have been properly set up by writing into location 0284 or 0285) to detect logic transitions on the most significant line of the A I/O port. Bits zero through five will always be read as zero.
0295	(R)	Same as 0285 except reading this location disables setting of the timer-count-down flag when the timer has counted down to zero.
029D	(R)	Same as 0285 except reading this location enables setting of the timer-count-down flag when the timer has counted down to zero.
0284	(W)	Enable setting of bit-seven-transition flag when line seven on the A I/O port goes from logic state one to logic state zero.
0285	(W)	Enable setting of bit-seven-transition flag when line seven on the A I/O port goes from logic state zero to logic state one.

0286	(W)	Disable setting of bit-seven-flag for a logic one to logic zero transition.
0287	(W)	Disable setting of bit-seven-flag for a logic zero to logic one transition. (It does not matter what value is written into locations 0284 through 0287.)
0294	(W)	Set timer count down value, begin counting down every microprocessor clock cycle, and disable setting of timer-count-down flag.
0295	(W)	Set timer count down value, begin counting down every eighth microprocessor clock cycle, and disable setting of timer-count-down flag.
0296	(W)	Set timer count down value, begin counting down every sixty-fourth microprocessor clock cycle, and disable setting of timer-count-down flag.
0297	(W)	Set timer count down value, begin counting down every 1024-th (decimal) microprocessor clock cycle, and disable setting of timer-count-down flag.
029C	(W)	Same as location 0294 except enable setting of timer-count-down flag.
029D	(W)	Same as location 0295 except enable setting of timer-count-down flag.
029E	(W)	Same as location 0296 excpet enable setting of timer-count-down flag.
029F	(W)	Same as location 0297 except enable setting of timer-count-down flag.

6.2 Game Controller and Switch Connections

In this section we discuss how the Atari game controllers (keyboard, joystick, and paddle) are interfaced to the microprocessor. The two controllers are connected to the Video Computer System via two nine pin connectors. When viewed from the back of the game, the pins on the connectors are numbered as follows:

Figure 10

The following abbreviations will be used in describing how the game controllers are connected:

LCn = Pin number "n" on left controller connector (n runs from one to nine).

RCn = Pin number "n" on right controller connector (n runs from one to nine).

PAn = RIOT I/O port A line number "n" (n runs from zero to seven).

PBn = RIOT I/O port B line number "n" (n runs from zero to seven).

6.2.1 Rear Connector Wiring

The two rear connectors are wired inside the game as follows:

Pin	Connected to
RC1	PA0
RC2	PA1
RC3	PA2
RC4	PA3
RC5	Display Generator control register at location 3A

RC6 Display Generator control register at location 3D

RC7 +5 Volts (note that the game is not capable of supplying more than 10 mA of current)

RC8 Power ground

RC9 Display Generator control register at location 3B

LC1 PA4

LC2 PA5

LC3 PA6

LC4 PA7

LC5 Display Generator control register at location 38

LC6 Display Generator control register at location 3C

LC7 +5 Volts (note that the game is not capable of supplying more than 10 mA of current)

LC8 Power ground

LC9 Display Generator control register at location 39

Rear connector pins RC5, RC6, RC9, LC5, LC6, and LC9 are each "connected to" a register in the Atari display generator (see Section 6.3). A program can read the display generator register to determine the voltage that is applied to the rear connector pin associated with the register. If the pin is grounded, a positive value will be read; if the pin is connected to +5 volts or left floating, a negative value will be read. It takes a couple of hundred microseconds for a change in voltage at a connector pin to be reflected in the value read from a control register. Rear connector pins LC6 and RC6 are electrically connected in a slightly different way than the other connector pins.

6.2.2 Console Switches Connections

The switches on the console of the Video Computer System are connected to I/O port B as follows:

Line	Connection
PB0	Game Reset switch. The PB0 line sees a logic zero when the switch is pressed.
PB1	Game Select switch. The PB1 line sees a logic zero when the switch is pressed.
PB2	
PB3	TV Type switch. When the switch is in the color position, line PB3 sees a logic one signal. This switch goes nowhere else.
PB4	
PB5	
PB6	Left Difficulty switch. Line PB6 sees a logic one signal when the switch is in the up position.
PB7	Right Difficulty switch. Line PB7 sees a logic one signal when the switch is in the up position.

6.2.3 Keyboard Controller Connections

Each button on a keyboard controller is an electrical switch that is connected between one line from I/O port A and one of the display controller registers. All the buttons in the same row (e.g. 1, 2, and 3) are connected to the same I/O port line. All the buttons in the same column (e.g. 1, 4, 7, and *) are connected to the same display generator register. The connections to the left (right) controller are shown in Figure 11.

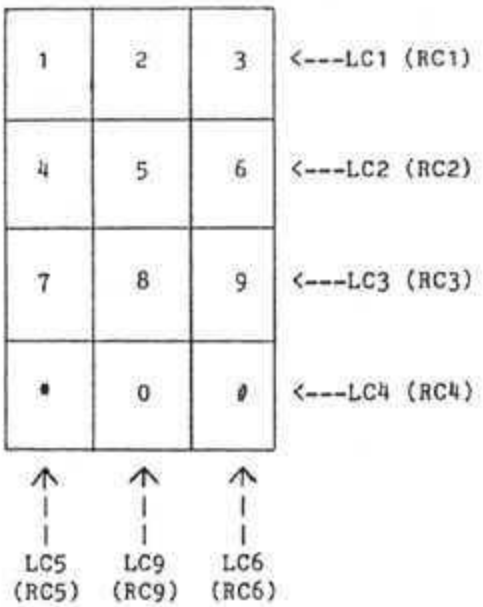


Figure 11

6.2.4 Joystick Controller Connections

In the description of the joystick controller connections, we will use the symbol "Cn" (where "n" runs from 1 to 9) to stand for the n'th pin on either the right or the left rear connector. If a joystick is held so that the red button is in the upper left corner, the following list describes the joystick operation:

1. Stick not pushed in any direction: Pins C1, C2, C3, and C4 are "floating" (that is, not electrically connected to anything).
2. Stick pushed forward: Pin C1 is connected to pin C8 (ground).
3. Stick pulled backwards: Pin C2 is connected to pin C8 (ground).
4. Stick pushed to the left: Pin C3 is connected to pin C8 (ground).
5. Stick pushed to the right: Pin C4 is connected to pin C8 (ground).
6. Red button not pushed: Pin C6 is floating.
7. Red button pushed: Pin C6 is connected to pin C8 (ground).

Pins C1 through C4 are connected to lines from RIOT I/O port A. When the I/O port lines are read into the computer, a floating line will show a value of one, while a grounded line will show a value of zero. The C6 pin is connected to one of the display generator control registers. If the pin is floating, the control register will read a negative value, while a grounded pin will read a positive value.

If the joystick is pushed diagonally, two pins are supposed to be grounded at the same time. For example, moving the stick back and to the left should connect both C2 and C3 to C8. In reality, however, some joysticks will work this way while others will not. You may have discovered this while playing games that attempt to use this feature.

6.2.5 Paddle Controller Connections

As in the joystick description, we use the symbol Cn to represent the n'th pin on either the left or right rear connector. The paddle controllers are wired as follows. Two controllers are connected to each rear connector. Each controller has a switch button and a dial. The dial is connected to a one-megaohm linear taper potentiometer which is at minimum resistance when the dial is turned completely clockwise. One of the controllers has its button wired so pressing it connects (the otherwise floating) C4 to C8 (ground). This controller's potentiometer is connected between pins C5 and C7 (+5 volts). Pressing the other controller's button connects pin C3 to C8, and its potentiometer is connected between pins C9 and C7.

6.3 Atari Display Generator

The Atari Display Generator is a custom built integrated circuit that, under the control of the 6502 microprocessor, can generate a sophisticated display on your TV screen, and can produce a wide variety of sounds from your TV speaker. There are actually two kinds of displays that the display generator can produce--a "low resolution" display (used by the MagiCard monitor for its TV display) and a "high resolution" display (used by some Atari game cartridges to make more detailed displays). The high resolution display is very complex; therefore, we can only give a brief description of how its control registers work.

For both kinds of displays, the display is generated one line at a time. Thus, to draw an object on the TV screen, a program must keep track of which scan line is being made on the TV screen, and then set the proper registers in the display generator to the proper value for each line on which the object appears. This is a tricky business which is one of the reasons why we have provided the DSPL subroutine to help you make displays. More information about synchronizing the display with your TV set is given in Section 6.5. More information about sound generation is given in Section 6.4.

As with the RIOT device, the display generator consists of a set of control registers, each with an address with which it is referenced. Each control register is either read only or write only. The control registers and their functions are as follows:

Address	(Read/Write)	Function
00	(W)	When bit #1 is set to one, start TV vertical sync period. When set to zero, end the vertical sync period.
01	(W)	When bit #1 is set to one, start TV vertical blanking period. When set to zero, end the vertical blank period.
02	(W)	When anything is written into this register, the microprocessor will stop executing instructions until the begining of the next TV horizontal scan line.
03	(W)	Unused
04	(W)	Controls the placement, repetition, and magnification of the high resolution masks at locations 1B and 1D. Writing zero into this location gives a single, unmagnified copy.
05	(W)	Same as 04 except for the masks at locations 1C, and 1E.
06	(W)	"Left color". Used for 1B,1D high resolution masks and for the left half of the low resolution display.
07	(W)	"Right color". Used for 1C,1E high resolution masks and for the right half of the low resolution display.
08	(W)	"Alternate color". Can be used for <u>both</u> halves of the low resolution display.
09	(W)	"Background color". This is the color of the screen where nothing is displayed.
0A	(W)	Bit #0 = 1 => reverse right half of low resolution display. Bit #1 = 1 => left and right colors used for low resolution display (otherwise, both sides are "alternate" color).
0B	(W)	Bit #3 = 1 => display 1B high resolution mask in reversed bit order.
0C	(W)	Same as 0B except for 1C high resolution mask.
0D	(W)	Most significant four bits (ordered from bit 4 through bit 7) are the beginning of the twenty-bit low resolution mask.
0E	(W)	Next eight bits (ordered from bit 7 through bit 0) of the low resolution mask.
0F	(W)	Last eight bits (ordered from bit 0 through bit 7) of the low resolution mask.

Each bit in the twenty-bit low resolution mask controls whether a corresponding position of the current TV scan line will be the low resolution color or the background color. The twenty-bit mask is actually displayed twice on each line, once on the right side of the TV screen and once on the left. However, the contents of the bit mask registers (0D, 0E, and 0F) can be changed by the program during the drawing of the line by the TV set. Thus, forty independently controllable line segments can be drawn on each scan line using the low resolution display (This is how the MagiCard monitor and the DSPL subroutine make TV displays.).

10 - 14 (W) A store into one of these locations positions the corresponding high resolution mask (1B - 1F) horizontally within a scan line as determined by the delay from the beginning of the scan until when the store is made.

15 (W) Sound channel #1, sound-type register (only the least significant four bits are used).

16 (W) Sound channel #2, sound-type register

17 (W) Sound channel #1, pitch register (only the least significant five bits are used).

18 (W) Sound channel #2, pitch register.

19 (W) Sound channel #1, volume register (only the least significant four bits are used).

1A (W) Sound channel #2, volume register.

1B (W) High resolution display mask. The bits written into this register define the shape of a high resolution display object (the most significant bit defines the left side of the object). The color of the object is set by storing into 06.

1C (W) A second high resolution display mask. The color is set by storing into 07.

1D (W) Setting bit #1 displays a short interval on the scan line (1/160-th of a scan line) in the color set by storing into 06.

1E (W) Setting bit #1 displays a short interval of the scan line (1/160-th of a scan line) in the color set by storing into 07.

1F (W) Setting bit #1 displays a short interval of the scan line (like 1C and 1D) but the color is always white.

20 - 24 (W) Sets the horizontal-position, shift-increment value (see description of locations 2A and 2B)

for the high resolution objects defined using locations 1B through 1F. The values stored here should range from -80 (a right shift of 8 places) to +70 (a left shift of 7 places). The total width of a TV scan line is 160 (decimal) high resolution display elements (each high resolution element is one-fourth the width of a low resolution element).

25 - 29 (W) Additional control of 1B through 1F high resolution objects.

2A (W) Writing into this location increments the horizontal position of <u>all</u> the high resolution objects by the amounts stored in their associated horizontal-position, shift-increment registers (locations 20 through 24).

2B (W) Storing into this location sets to zero the values of <u>all</u> the horizontal-position, shift-increment registers (locations 20 through 24).

30 - 37 (R) Display generator status information.

38 - 3D (R) Analogue input from controller connections (see Section 6.2). A negative value indicates an input voltage above threshold voltage.

6.4 Details of Sound Generation

The Atari display generator control registers between locations 15 and 1A can be used to produce sounds from the TV speaker. There are actually two independently controllable sound "channels"-- each of which has a pitch register, a volume register, and a sound-type register. The sound that comes out of the TV speaker is a mixture of the sounds that have been selected from the two channels.

The value stored into the pitch register (location 17 for the first sound channel, location 18 for the second sound channel) specifies the pitch of the sound. Storing a value of 00 produces the highest pitch; storing a value of 1F produces the lowest pitch. The value stored into the volume register (location 19 for the first sound channel, location 1A for the second sound channel) specifies the volume of the sound. Storing a value of 00 produces silence; storing a value of 0F produces the loudest possible sound. The volume of the two channels is not totally independent-- as one gains volume, the other loses volume. The value stored into the sound-type register (location 15 for the first sound channel, location 16 for the second sound channel) specifies the type of sound to be made. Values from 00 through 0F may be stored into these registers. Of the sixteen possible values, two (00 and 0B) produce no sound. The other values produce a variety of sounds that are difficult to describe. The easiest thing to do is to experiment by storing various values into the sound registers and listening to what happens ! One final comment--because the 6502 executes instructions at a very fast rate compared to the length of the sounds that are produced, "real-time" manipulation of the sound control registers can produce a wide variety of effects.

6.5 Details of TV Display Generation

The most convenient way to make a TV display is to use the DSPL monitor subroutine described in Section 7.1. A program can also make a display on the TV screen by storing into the display controller registers in a precisely timed manner. The rest of Section 6.5 describes how to make a display without using the DSPL subroutine. Example 4 in Chapter 2 gives an example of how the procedure described below can be used in a program.

6.5.1 Basic Display Generation Procedure

Any program that makes a stable display on the TV screen must repeat a basic set of operations once every one-sixtieth of a second (the time it takes a TV set to draw one frame). The time it takes for the computer to execute this "basic loop" can be precisely controlled by using the timer contained in the RIOT device (see Section 6.1) in combination with the display controller register at location 02. If the basic loop takes more or less than one-sixtieth of a second to execute, the TV picture will wiggle slightly and may roll vertically.

The operations in the basic loop are as follows:

1. Start the vertical-blanking interval.
2. Start the vertical-sync interval.
3. End the vertical-sync interval.
4. End the vertical-blanking interval.
5. Set up each line of the TV picture as the line is being drawn by the TV set.
6. Go back to Step 1.

Each of the steps in the basic loop will now be discussed in more detail.

6.5.2 Start the Vertical-Blanking and Vertical-Sync Intervals

The vertical-blanking and vertical-sync time intervals are begun at the same time. The operations necessary to begin the blanking and sync intervals are as follows:

1. Store something (anything) into the display controller register at location 02.
2. Store a byte with bit #1 set into the display controller register at location 01.
3. Store a byte with bit #1 set into the display controller register at location 00.

The TV set responds to these operations by:

1. Turning off the "electron gun" that it uses to draw the picture on the TV screen. This is done to keep from messing up the TV picture when the gun moves from the bottom of the screen to the top.
2. Starting to move the gun from the bottom of the current TV frame to the top of the next frame.

6.5.3 End the Vertical-Sync Interval

The vertical-sync interval must have a short (a few hundred microseconds) fixed duration. The RIOT timer function can be used to keep track of how much time is left before the vertical-sync interval should be ended. A good procedure to follow is:

1. Immediately after the vertical-sync interval has been started, store the value 2A into the RIOT timer control register at location 0295. This initializes the timer to a value of 2A and starts it counting down toward zero.
2. At this point, there is time for about eighty average 6502 instructions to be executed before the timer reaches zero. Your program can use this time to perform any calculations it needs to do.
3. After the program has done its calculations (if there were any), it should read location 0284 (the time remaining in the timer count down). If the value read is not zero, the program should loop back and read 0284 again.
4. When the program reads a value of zero from location 0284, it

should end the vertical-sync interval by writing a zero into the display controller registers at locations 02 and 00 (in that order).

By the end of the vertical-sync interval, the electron gun (still turned off because the vertical-blanking interval has not yet ended) has reached the top of the screen and is beginning to draw a new frame of the TV picture.

6.5.4 End the Vertical-Blanking Interval

Unlike the vertical-sync interval, the length of the vertical-blanking interval can be made as long as desired. Once the vertical-sync interval has ended, continuing the vertical-blanking interval causes the lines of the picture at the top of the screen to be blank. Because the program will often be very busy forming the display after the vertical-blanking has ended, much of the computing done by a program must be done before vertical-blanking is ended. If your program does a lot of computation, the vertical-blanking interval can be extended (thus moving the first nonblank line of the picture further down the screen, and reducing the total number of nonblank lines available for your display).

If you look at the display produced by different game cartridges, you will see the blank area at the top of the screen is larger for some games than for others. The games with a larger blank area usually need the extra time for computations. At least one game (chess) shows a single-color totally-unsynchronized display while it is calculating its next move. This gives the program 100% of the time for performing calculations.

A normal length vertical blanking period can be ended as follows:

1. Immediately after the vertical-sync interval has been ended, write the value 24 into location 0296 (the RIOT's count down timer).

2. At this point, there is time for about five hundred and fifty average length 6502 instructions before the timer reaches zero.

3. After the program has performed its calculations, it should read location 0284 (the time remaining in the timer count down). If the value read is not zero, the program should loop back and read 0284 again.

4. When the program reads a value of zero from location 0284, if should end the vertical-blanking interval by writing a zero into the display controller registers at locations 02 and 01 (in that order).

6.5.5 Set Up Each Line of the TV Picture

After the vertical-blanking interval has been ended, the TV's electron gun is turned on, and the horizontal lines drawn by the gun will be visible on the TV screen. If the above recommended value for the length of the vertical-blanking interval is used (i.e. starting the RIOT timer by storing 24 into location 0296), the TV set must draw two hundred and twenty-eight lines (E4 hex) on the screen before the start of the next vertical-sync interval. (However it is very possible that the last ten or twenty lines drawn will end up below the bottom of your TV screen.) Your program can make its display on the screen by telling the display generator what to place on each line of the display.

The program should start out by writing any value into the display controller register at location 02. This causes the 6502 microprocessor to stop executing instructions until the TV set is just beginning to draw the next line on the screen. The instructions immediately following the write into 02 should write into the display generator registers needed to produce the display that you want to make.

For example, if you are using the low resolution display feature (see Section 6.3), you must quickly write into locations 0D, 0E, and

OF to tell the display generator what pattern is to be displayed on the left half of the TV screen. If you want to place a new pattern on the right hand side of the screen, you must then reload the same three locations--after the bits loaded into each register have been displayed on the left side of the screen, but before they are displayed on the right side. The time it takes the TV to draw a line is sufficient for about eighteen 6502 instructions to be executed (or more precisely, exactly seventy-six microprocessor clock cycles).

6.5.6 Additional Comments

If more time is needed for computing between TV frames, the vertical-blanking interval can be lengthened and the number of visible lines in the display reduced. One line is removed from the visible display for every seventy six microprocessor clock cycles added to the vertical-blanking interval. If the picture on your TV "wiggles" slightly, the time the program is using to make each TV frame is either a little too long or a little too short. Adjust either the number of lines in the visible part of the picture, or adjust the length of the vertical-blanking interval until the picture is stable. You may find it helpful to dump (using the "Instruction Dump" feature of the MagiCard) the MagiCard monitor display subroutine "DSPL" (see Section 7.1) to see an example of how a program can generate a TV display.

CHAPTER 7

MagiCard Monitor Subroutines

In this section, we describe a number of useful subroutines that are part of the MagiCard Monitor program. These subroutines are contained in read-only memory so it is impossible for them to be overwritten by a programming error. The following information is provided for each subroutine:

1. Subroutine Name--A four letter name for the subroutine. The name is given for convenience only and has no meaning to the monitor program.
2. Subroutine Address--The four-hex-digit absolute address of the subroutine.
3. Subroutine Description--A short description of what the subroutine does.
4. Subroutine Calling Procedure--What input arguments the subroutine has and where they are to be placed before calling the subroutine.
5. Subroutine Outputs--What output arguments are produced by the subroutine and where they are to be found.
6. Subroutine Side Effects--What memory locations and registers (other than those registers and memory locations that are used for the output parameters) are modified by calling the subroutine.
7. Example--An example of how the subroutine can be used.

7.1 Refresh TV Display and Read Keyboard Controllers

Name: DSPL

Address: FA6F

Description: This subroutine makes a display on the TV screen (for one

"frame" lasting one-sixtieth of a second) and determines if any of the keys on the keyboard controllers were pushed during that time. Any program that is using DSPL to display information on the TV screen must call DSPL at least once each sixtieth of a second to maintain a steady picture. A significant amount of the allowed time between calls to DSPL is taken up by the subroutine itself before it returns. This leaves enough time for the user program to execute about 550 (decimal) average length instructions between calls.

For display purposes, the TV screen is divided into a set of equal-sized rectangular regions called "picture elements" or "pixels". There are forty-two rows of pixels containing forty pixels each. Each pixel can be individually turned "on" (i.e. be seen as a small colored rectangle on the TV screen) or turned "off" (i.e. be seen as a small black rectangle on the TV screen). Furthermore, each pixel has associated with it a unique memory bit (called the "pixel-bit"). When the DSPL subroutine is called, it uses this pixel-bit to control whether or not the pixel will be displayed as "on" or "off". If the pixel-bit is set to one, the corresponding pixel will be "on". If the pixel-bit is set to zero, the corresponding pixel will be "off".

The pixel-bits are stored in memory locations F000 through F0D1. Because of the complicated way the Atari Video Computer System makes a TV display (see Section 6.3), the pixel-bits are stored in memory in a funny way. The subroutine CALP (see Section 7.3) can be used to find the location of the pixel-bit for any desired pixel.

Calling Procedure:

The subroutine is called via the instruction "JSR [A] FA6F". DSPL has no input parameters but it does depend on proper initialization of the Video Computer System Display Generator. This initialization is made by the MagiCard Monitor program whenever a reset is made (e.g. when the system is powered up or when a BRK instruction is executed). Only the colors of the display are likely to be usefully changed in a user program. The background color is set by storing a value into location 09. The pixel's color is set by storing a value into location 08. More details about color control can be found in Section 6.3 in which the Display Generator is discussed.

Outputs:

location 82: indicates if a new key was pressed on the left keyboard controller. If the contents of this location are negative, no new key has been pressed. If the contents are positive, it is the value of the key that was pressed (Pressing the "*" key returns a value of 0A, and pressing the "#" key returns a value of 0B). If a key is pressed and held down, it will only be reported on the first call to DSPL unless it is released and pressed again.

location 83: indicates if a new key was pressed on the right keyboard controller.

location 8A: indicates if a key is currently being pressed on the left keyboard controller. The contents of this location are set in the same way as location 82 except that if a key is pressed and held down it will be reported on each call to DSPL until it has been released (unlike location 82 where it would be reported only once).

location 8B: indicates if a key is currently being pressed on the

right keyboard controller.

Side Effects:
The accumulator, X index register, Y index register and condition codes are all altered.
DSPL uses the timer function of the 6532 RIOT device (see Section 6.1.3). The timer can be read by a user program; however, if it is altered between calls to DSPL, TV synchronization will be lost.
DSPL uses the "A" I/O port function of the 6532 RIOT device. The I/O port can be used by a user program, but DSPL will modify it each time it is called.
If the "Game Reset" switch on the Video Computer System console is depressed when DSPL is called, a BRK instruction will be executed forcing a reset of the MagiCard Monitor (see Section 7.15.1). This is a useful way to terminate the execution of a program that calls DSPL. The I-flag in the processor status register can be set to prevent the reset from occurring.

7.2 Store to Program Memory

Name: STPM

Address: FFF7

Description: Store the contents of the accumulator into a memory location in program memory (i.e. between address F000 and F3FF). The address at which the accumulator is to be stored is the sum of a specified sixteen-bit absolute address (called the "base address") and the contents of the Y index register (called the "address offset"). Warning: The sum of the Y register and the low order 8 bits of the base address should not exceed FF.

Calling Procedure:
1. Place the least significant byte of the base address into memory location 8C.
2. Add four to the most significant byte of the base address and place the result into memory location 8D (i.e., if the base address is F000, zero should be placed into location 8C and, F4 should be placed into location 8D).
3. Place the value of the address offset into the Y index register.
4. Place the value to be stored into the accumulator.
5. Execute the instruction "JSR [A] FFF7".

Outputs: None

Side Effects: None

Example: The following program fragment stores a one into memory location F106:

```
LDA  [I]  00      clear the accumulator
STA  [Z]  8C      clear location 8C
LDA  [I]  F5      load accumulator with F1 + 4
STA  [Z]  8D      put value F5 into location 8D
```

```
LDY  [I]  06        place offset value of 6 into Y
LDA  [I]  01        set accumulator to one
JSR  [A]  FFF7      call subroutine STPM
```

7.3 Calculate Address and Mask for Plotting

Name: CALP

Address: FFBE

Description: Calculate an address and bit mask to use for plotting. This subroutine aids MagiCard users in making displays on the TV screen. As explained in Section 7.1, the display generating subroutine produces a display consisting of pixels each with an associated pixel-bit in memory. The CALP subroutine takes as input the pixel's position on the TV screen (i.e., its row number and column number) and returns as output the information needed to set (or clear) the pixel bit.

Calling Procedure:

1. Place the Horizontal position of the pixel into memory location 8E. The horizontal position is a number between zero and thirty-nine (27 hex) where a value of zero specifying the left side of the screen and a value of thirty-nine specifying the right side.
2. Place the vertical position of the pixel into memory location 8F. The vertical position is a number between zero and forty-one (29 hex) where a value of zero specifies the top of the screen and a value of forty-one specifies the bottom.
3. Execute the instruction "JSR [A] FFBE".

Outputs:

Y index register: when added to F000, the value contained in the Y index register gives the address of the byte containing the pixel-bit.

Accumulator: has a bit set that corresponds to the pixel-bit within the byte.

Side Effects:

N,Z,C,and V-flags are altered
memory location 90 is altered

Examples: The following examples assume that memory locations 8C and 8D have been previously filled with 00 and F4 respectively, and that memory locations 8E and 8F contain the horizontal and vertical position of the pixel being worked with.

Example 1: turn on the selected pixel-bit

```
JSR  [A]  FFBE      calculate address and bit mask
ORA  [Y]  F000      "OR" new bit with others
JSR  [A]  FFF7      store results (with subroutine
                    STPM)
```

Example 2: turn off the selected pixel-bit

right keyboard controller.

Side Effects:

The accumulator, X index register, Y index register and condition codes are all altered.

DSPL uses the timer function of the 6532 RIOT device (see Section 6.1.3). The timer can be read by a user program; however, if it is altered between calls to DSPL, TV synchronization will be lost.

DSPL uses the "A" I/O port function of the 6532 RIOT device. The I/O port can be used by a user program, but DSPL will modify it each time it is called.

If the "Game Reset" switch on the Video Computer System console is depressed when DSPL is called, a BRK instruction will be executed forcing a reset of the MagiCard Monitor (see Section 7.15.1). This is a useful way to terminate the execution of a program that calls DSPL. The I-flag in the processor status register can be set to prevent the reset from occurring.

7.2 Store to Program Memory

Name: STPM

Address: FFF7

Description: Store the contents of the accumulator into a memory location in program memory (i.e. between address F000 and F3FF). The address at which the accumulator is to be stored is the sum of a specified sixteen-bit absolute address (called the "base address") and the contents of the Y index register (called the "address offset"). Warning: The sum of the Y register and the low order 8 bits of the base address should not exceed FF.

Calling Procedure:

1. Place the least significant byte of the base address into memory location 8C.
2. Add four to the most significant byte of the base address and place the result into memory location 8D (i.e., if the base address is F000, zero should be placed into location 8C and, F4 should be placed into location 8D).
3. Place the value of the address offset into the Y index register.
4. Place the value to be stored into the accumulator.
5. Execute the instruction "JSR [A] FFF7".

Outputs: None

Side Effects: None

Example: The following program fragment stores a one into memory location F106:

```
LDA  [I]  00      clear the accumulator
STA  [Z]  8C      clear location 8C
LDA  [I]  F5      load accumulator with F1 + 4
STA  [Z]  8D      put value F5 into location 8D
```

```
LDY  [I]  06      place offset value of 6 into Y
LDA  [I]  01      set accumulator to one
JSR  [A]  FFF7    call subroutine STPM
```

7.3 Calculate Address and Mask for Plotting

Name: CALP

Address: FFBE

Description: Calculate an address and bit mask to use for plotting. This subroutine aids MagiCard users in making displays on the TV screen. As explained in Section 7.1, the display generating subroutine produces a display consisting of pixels each with an associated pixel-bit in memory. The CALP subroutine takes as input the pixel's position on the TV screen (i.e., its row number and column number) and returns as output the information needed to set (or clear) the pixel bit.

Calling Procedure:

1. Place the Horizontal position of the pixel into memory location 8E. The horizontal position is a number between zero and thirty-nine (27 hex) where a value of zero specifying the left side of the screen and a value of thirty-nine specifying the right side.
2. Place the vertical position of the pixel into memory location 8F. The vertical position is a number between zero and forty-one (29 hex) where a value of zero specifies the top of the screen and a value of forty-one specifies the bottom.
3. Execute the instruction "JSR [A] FFBE".

Outputs:

Y index register: when added to F000, the value contained in the Y index register gives the address of the byte containing the pixel-bit.

Accumulator: has a bit set that corresponds to the pixel-bit within the byte.

Side Effects:

N,Z,C,and V-flags are altered
memory location 90 is altered

Examples: The following examples assume that memory locations 8C and 8D have been previously filled with 00 and F4 respectively, and that memory locations 8E and 8F contain the horizontal and vertical position of the pixel being worked with.

Example 1: turn on the selected pixel-bit

```
JSR  [A]  FFBE    calculate address and bit mask
ORA  [Y]  F000    "OR" new bit with others
JSR  [A]  FFF7    store results (with subroutine
                  STPM)
```

Example 2: turn off the selected pixel-bit

```
JSR  [A]  FFBE      calculate the address and bit mask
EOR  [I]  FF        complement bit mask
AND  [Y]  F000      turn off selected bit
JSR  [A]  FFF7      store the result
```

Example 3: reverse the state of the pixel-bit (turn it off if it is on, or turn it on if it is off)

```
JSR  [A]  FFBE      calculate the address and bit mask
EOR  [Y]  F000      complement the pixel-bit bit
JSR  [A]  FFF7      store the result
```

Example 4: test if the pixel is on

```
JSR  [A]  FFBE      calculate the address and bit mask
AND  [Y]  F000      this will clear the Z-flag if the
                    pixel's bit is on
BNE  (somewhere)    branch if the pixel is on
```

7.4 Place One Character Into The Display

Name: ONEC

Address: FB67

Description: Set the pixel-bits needed to plot a character. Section 7.1 explains how the state of each pixel displayed on the TV screen (by the TV display subroutine) is controlled by the value of its associated pixel-bit in memory. By setting the proper combination of pixel-bits, it is possible to display characters (letters and numbers) on the screen as well. Each character occupies a rectangle four pixels wide and six pixels high (including the "off" pixels that separate adjacent characters). The TV screen can hold seven lines of ten characters each. ONEC is called to set the pixel bits necessary (and clear the pixel-bits that are not necessary) to place a character onto the TV screen at the selected position.

Calling Procedure:

1. Place the line number of the character position on the screen to be filled into location 8F. The line number is a number between zero (specifying the top line of characters) and six (specifying the bottom line of characters).
2. Place the character number of the character position on the screen to be filled into location 8E. The character number is a number between zero (specifying the left side of the screen) and nine (specifying the right side of the screen). Neither the line nor character numbers are checked to see if they are legal values. The use of illegal values may produce various problems with the display or with the user program.
3. Place the ASCII code (see Appendix B) for the character to be displayed into memory location 98 + character number (e.g., when displaying the character at character number 9, place the ASCII code into location 98 + 9 = A1).

4. Execute the instruction "JSR [A] FB67".

Outputs: None

Side Effects:
The accumulator, X index register, Y index register, and condition codes are altered.
Locations 82,83 and 90 through 97 are modified.
This routine cannot be called more than twice between calls to DSPL if loss of TV synchronization is to be avoided.

Example: The following program fragment sets the pixel-bits required to display the letter "Z" in the bottom right-hand corner of the TV screen:

```
LDA  [I]  09      load 09 into accumulator
STA  [Z]  8E      store into location 8E
LDA  [I]  06      load 06 into accumulator
STA  [Z]  8F      store into location 8F
LDA  [I]  5A      load ASCII for "Z" into accumulator
STA  [Z]  A1      store into A1
JSR  [A]  FB67    call ONEC
```

7.5 Place a Line of Characters Into the Display

Name: DOLN

Address: FCC8

Description: Set the pixel-bits necessary to display an entire line of characters. This subroutine is similar to subroutine ONEC except it works with an entire line of characters---not just one.

Calling Procedure:
1. Place the line number into location 8F. The line number is a value from zero (specifying the top of the TV screen) to six (specifying the bottom of the TV screen).
2. Place the ASCII codes (see Appendix B) for the characters to be displayed into locations 98 through A1 (location 98 is for the left most character on the screen).
3. Execute the instruction "JSR [A] FCC8".

Outputs: None

Side Effects:
The accumulator, X index register, Y index register, and condition codes are altered.
DOLN calls the DSPL subroutine to keep the display going; however, it destroys the contents of locations 8A and 8B. To find out the value of the key currently pressed on each keyboard controller, it is necessary to call DSPL after the return from DOLN.
Upon return from a call to DOLN, about half the time between calls to DSPL necessary for TV synchronization will have elapsed.

7.6 Fill the Line Buffer With Blanks

Name: BFIL

Address: FDAC

Description: This subroutine fills locations 98 through A1 with the ASCII code for a blank character (20). This range of memory locations is used by the DOLN subroutine to set the pixel-bits necessary to display a line of characters. It is often desired to display a line of text that contains blanks in many of the ten character positions. To do this, you can call the BFIL subroutine, place the ASCII values for the nonblank characters into the proper places, and the call the DOLN subroutine.

Calling Procedure

Execute the instruction: JSR [A] FDAC

Outputs: None

Side Effects

The accumulator is set to 20.
The X index register is set to zero.
The N-flag is clear.
The Z-flag is set.

7.7 Scroll the Pixel-bits Up One Character Line

Name: SCRL

Address: FD8D

Description: This subroutine modifies the pixel-bits such that when the DSPL subroutine is called again, each pixel on the TV screen will be moved up 6 pixel lines (or equivalently, one line of characters). The former top six rows of pixel-bits are lost and the new bottom six lines of pixel-bits will contain a random bit pattern. It is expected that the DOLN subroutine will be called to set the pixel-bits for the bottom of the screen with the values needed to display a new line of text. This subroutine is used by the MagiCard Monitor program when it performs the hex dump and instruction dump functions.

Calling Procedure:

Execute the instruction "JSR [A] FD8D"

Outputs: None

Side Effects:

The accumulator is set to 20.
The X and Y index registers are set to zero.
The condition codes are altered.
Memory locations 98 through A1 are set to 20 (ASCII blanks).
Memory location 8F is set to the value 06.
SCRL calls the DSPL subroutine three times during its execution in order to maintain the TV picture synchronization. The

keyboard data returned by DSPL is ignored so the record of a new key being pressed will be lost if it occurs during the call to SCRL. In practice, however, the call to SCRL is short enough that the loss of key press data is almost never noticeable.
Note that SCRL leaves things very well set up to set the pixel-bits for a new line of text by calling DOLN.

7.8 Convert a Byte of Data Into ASCII

Name: NUMC

Address: FCB1

Description: This subroutine takes as input a one byte number and produces as output the two ASCII code values that are the hexadecimal representation of the number. The NUMC subroutine can be used in conjunction with the ONEC (or DOLN) subroutine and the DSPL subroutine to display the hexadecimal representation of a number on the TV screen.

Calling Procedure:

1. Place the value to be converted into the accumulator
2. Place into the X index register the memory location (with respect to location 98) where the first of the two ASCII code values is to be place. That is, if the X index register contains a zero, the first ASCII code value will be placed into location 98. If the X index register contains an eight, the first ASCII code value will be placed into location 98 + 8 = A0. The range of memory locations from 98 through A1 is used by the ONEC (or DOLN) subroutine as part of the character display procedure.
3. Execute the instruction "JSR [A] FCB1".

Outputs:

The memory locations at the addresses 98 + X index register and 99 + X index register are filled with the ASCII code values representing the hexadecimal value for the input value.
The X index register is incremented by two (so that another call to NUMC will fill the next two memory locations with ASCII code values).

Side Effects:

The accumulator is set to the ASCII code value for the least significant part of the input value.
The C and Z-flags are zero.
The N and V-flags are altered.

7.9 Convert Half a Byte of Data Into ASCII

Name: ENCL

Address: FCBA

Description: This subroutine is identical with subroutine NUMC with the following two exceptions:

1. Only the lower four bits of the accumulator are converted into ASCII code.

2. The X index register is incremented by one instead of by two.

7.10 Accumulate Hex Digits

Name: PSHD

Address: FD7D

Description: This subroutine is used to accumulate a four-hex-digit number, one hex digit at a time. Each time PSHD is called, the contents of memory locations 80 and 81 are modified as follows:

1. The upper-four bits of location 81 are discarded.

2. The lower-four bits of location 81 are moved into the upper-four bits of location 81.

3. The upper-four bits of location 80 are moved into the lower-four bits of location 81.

4. The lower-four bits of location 80 are moved into the upper-four bits of location 80.

5. The lower-four bits of the accumulator are moved into the lower-four bits of location 80.

The most useful way to think of the action of this subroutine is as a "shift" of the four-digit hex number contained in locations 80 and 81. The most-significant hex digit (the upper-four bits in location 81) is lost, and a new least-significant hex digit is obtained from the lower-four bits of the accumulator. The MagiCard monitor program uses this subroutine to accumulate the hex digits entered with the keyboard controller (the contents of locations 81 and 80 are displayed by the monitor on the upper right-hand corner of the TV screen).

Calling Procedure:
1. Place the value of the new hex digit into the accumulator
2. Execute the instruction "JSR [A] FD7D".

Outputs: None

Side Effects:
The accumulator contains the value of the hex digit shifted out of the upper-four bits of location 81.
The X index register is set to zero.
The Z-flag is set.
The N-flag is clear.

The C-flag is altered.

7.11 Add One to "Fetch Address" and Compare to "Cend Address"

Name: INCB

Address: FC9D

Description: One is added to the sixteen-bit value of the "Fetch Address" (least-significant byte in location 86, most-significant byte in location 87) and the result is compared to the value of the sixteen-bit value of the "Cend Address" (least-significant byte in location 8C, most-significant byte in location 8D).

Calling Procedure:
Execute the instruction "JSR [A] FC9D".

Outputs:
The Z-flag is set if the "Fetch Address" equals the "Cend Address" (after one has been added to the "Fetch Address").

Side Effects:
The accumulator, C-flag, V-flag, and N-flag are altered.

7.12 Add Accumulator to "Fetch Address" and Compare

Name: BMPB

Address: FC9F

Description: The same as subroutine INCB except the contents of the accumulator is added to the "Fetch Address" instead of one. (Note: The value contained in the accumulator is treated as an eight-bit unsigned number. For example, if the accumulator contains the value "FF", then two hundred and fifty-five will be added to the "Fetch Address".)

7.13 Determine Instruction Name

Name: INSN

Address: FC83

Description: This subroutine accepts as input one of the possible instruction opcode values, and returns as output the position of the three-ASCII-character instruction name within a table of instruction names.

Calling Procedure:

1. Place the value of the instruction opcode into the accumulator.
2. Execute the instruction "JSR [A] FC83".

Outputs:
The accumulator contains the position (from zero to sixty) of the instruction name within a table of instruction names. The instruction-name table begins at memory location F9A0. Each entry in the table is three bytes long and contains the ASCII character codes for the instruction name. If the accumulator returns with a value of zero, the input accumulator value was not a legal 6502 opcode. In this case, the associated table entry (entry zero) contains the nonexistent instruction name "ILL" indicating an illegal instruction.

Side Effects:
The Z, C, V, and N-flags are altered.
Memory location 93 is altered.

7.14 Determine Instruction Addressing Mode

Name: ADTY

Address: FC4F

Description: This subroutine accepts as input one of the possible instruction opcodes,and returns as output the position of the two-ASCII-character addressing mode name within a table of opcode names.

Calling Procedure:
1. Place the value of the instruction opcode into the accumulator.
2. Execute the instruction "JSR [A] FC4F"

Outputs:
The accumulator contains the position (from zero to eleven) of the addressing mode name within a table of addressing-mode names. The table begins at memory location FA57. Each entry in the table is two bytes long and contains the ASCII character codes for the addressing-mode name. The addressing-mode names are the same as those used by the MagiCard monitor instruction-dump function. A negative value in the accumulator indicates that the input accumulator value was not a legal 6502 opcode.

Side Effects:
1. The Z and N-flags are altered.
2. Memory location 93 is altered.

7.15 Determine Instruction Length

Name: INLN

Address: FC3D

Description: This subroutine accepts as input one of the possible addressing-mode table locations (as returned by the ADTY subroutine), and returns as output the number of bytes in the instruction.

Calling Procedure:

1. Place the value of the addressing-mode table position into the accumulator. (The Z-flag and N-flag values must reflect the value in the accumulator.)
2. Execute the instruction "JSR [A] FC3D"

Outputs:

The accumulator contains the number of bytes in the instruction (including the opcode byte).

Side Effects:

The Z, N, V, and C-flags are all altered.

7.16 Other Monitor Addresses

In this section, we describe some locations within the monitor program that, while they are not subroutines, are of potential interest. Some of these locations can be jumped to by a user program to cleanly go back to the monitor program. Other of the locations are given as a guide to those who wish to examine in more detail the monitor program itself by displaying the monitor program's instructions via the MagiCard "Ins Dump" function.

7.16.1 Reset Entry Point

Name: RSET

Address: FE5F

The microprocessor starts executing instructions at this location whenever it is powered up or when a BRK instruction is executed (and the I-flag in the processor status register is clear). The following operations are then performed:

1. The I-flag in the processor status register is cleared.
2. The D-flag in the processor status register is cleared (the monitor program will not work in decimal mode).
3. The stack-pointer register is set to FF.
4. Memory locations 80 through A9 are set to zero.
5. The display controller is initialized.
6. All the display pixel-bits are cleared (memory locations F000 through F0D1).

7. I/O port B of the RIOT is set to input mode.

8. The DOLN subroutine is called to set the pixel-bits needed to display the message "CMX-2" on the screen.

9. The main program of the MagiCard monitor is entered (see the description of MAIN in the next section).

A user program can also jump to this location if it wishes to go back to the MagiCard monitor program.

7.16.2 Beginning of Monitor Main Program

Name: MAIN

Address: FE96 This is the beginning of the main program for the MagiCard monitor. A user program can jump to this location if it is desired to return to the monitor program without performing the initialization operations that would be made by a jump to RSET. Because a jump to MAIN bypasses these initialization operations, a jump to MAIN will cleanly pass control to the monitor program only if:

1. The stack-pointer register is set to FF.

2. The display controller is reasonably well initialized.

A successful pass of control to the monitor program does not clear the screen but will cause a single blinking pixel to appear on the TV screen. If this does not happen, it is possible that the user program has set the color of the display to black. This problem can be cured by storing a number (such as 55) into memory location 08 (this sets the color of the display). If storing into 08 does not help, you can try to restart the monitor by pressing the "Game Reset" switch. This will perform the normal monitor reset sequence, but it is better than powering down the Video Computer System. Finally, if all else fails, you can regain control of the computer by turning the power off briefly.

7.16.3 Disassembler Program

Name: DISA

Address: FDB6 This is the beginning of the "instruction dump" portion of the MagiCard monitor program (executed when the "Ins Dump" key is pressed).

7.16.4 Cassette Write Program

Name: CWRT

Address: FCFA This is the beginning of the "cassette write" portion of the MagiCard monitor program (executed when the "Cwrite" key is pressed).

7.16.5 Cassette Read Program

Name: CRED

Address: FD34 This is the beginning of the "cassette read" portion of the MagiCard monitor program (executed when the "Cread" key is pressed).

7.16.6 Hex Dump Program

Name: HEXD

Address: FE38 This is the beginning of the "hexidecimal dump" portion of the MagiCard monitor program (executed when the "Hex Dump" key is pressed).

7.16.7 Start User Program Execution

Name: GO

Address: FE58 This is the beginning of the "run user program" portion of the MagiCard monitor program (executed when the "Run" key is pressed).

7.16.8 Relative Address Calculation

Name: RELC

Address: FF51 This is the beginning of the "calculate relative address" portion of the MagiCard monitor program (executed when the "Relc" key is pressed).

APPENDIX A

HEXADECIMAL DECIMAL CONVERSION TABLE

First Hex Digit	Second Hex Digit 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

To find the hex equivalent of a decimal number, first find the decimal number in the chart, and then read off the first and second hex digits from the row and column the decimal number is found in. To find the decimal equivalent of a hex number, locate the proper row and column pointed to by the first and second hex digits, then read off the decimal entry. For example, hex AB is decimal 171; decimal 212 is hex D4.

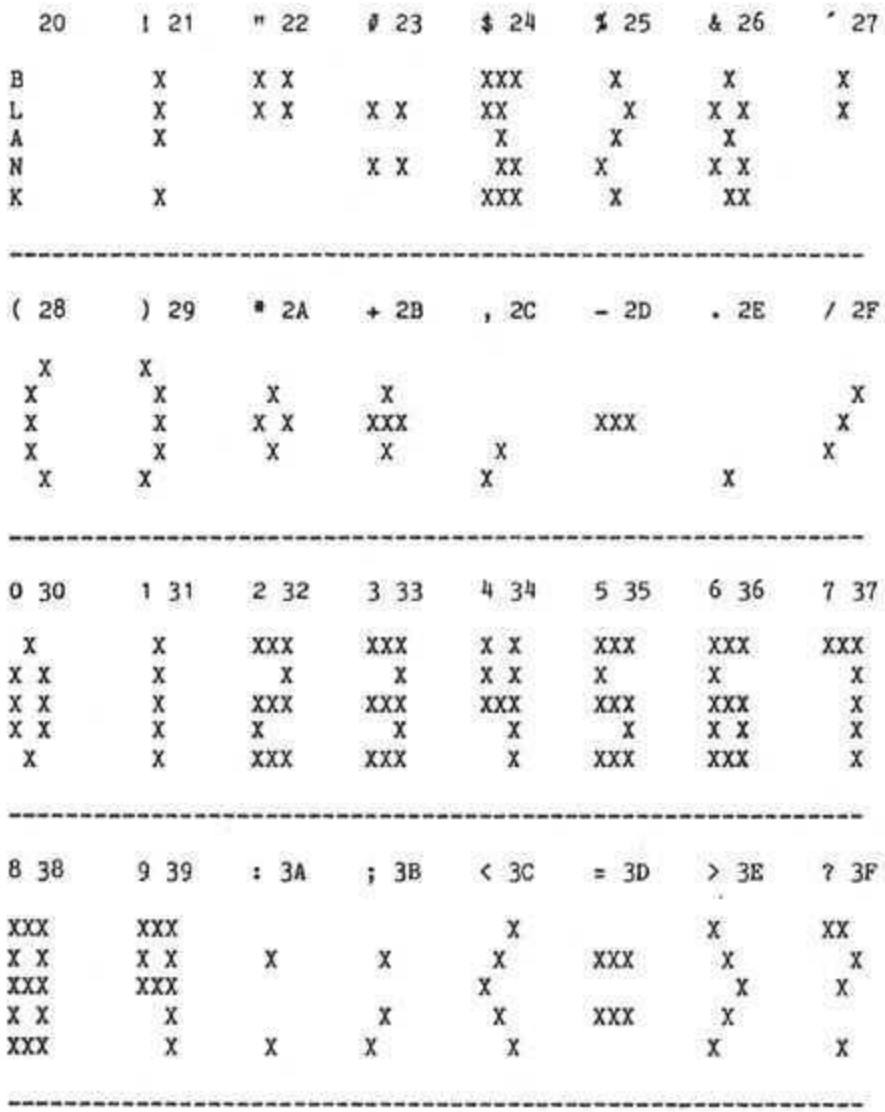
APPENDIX B

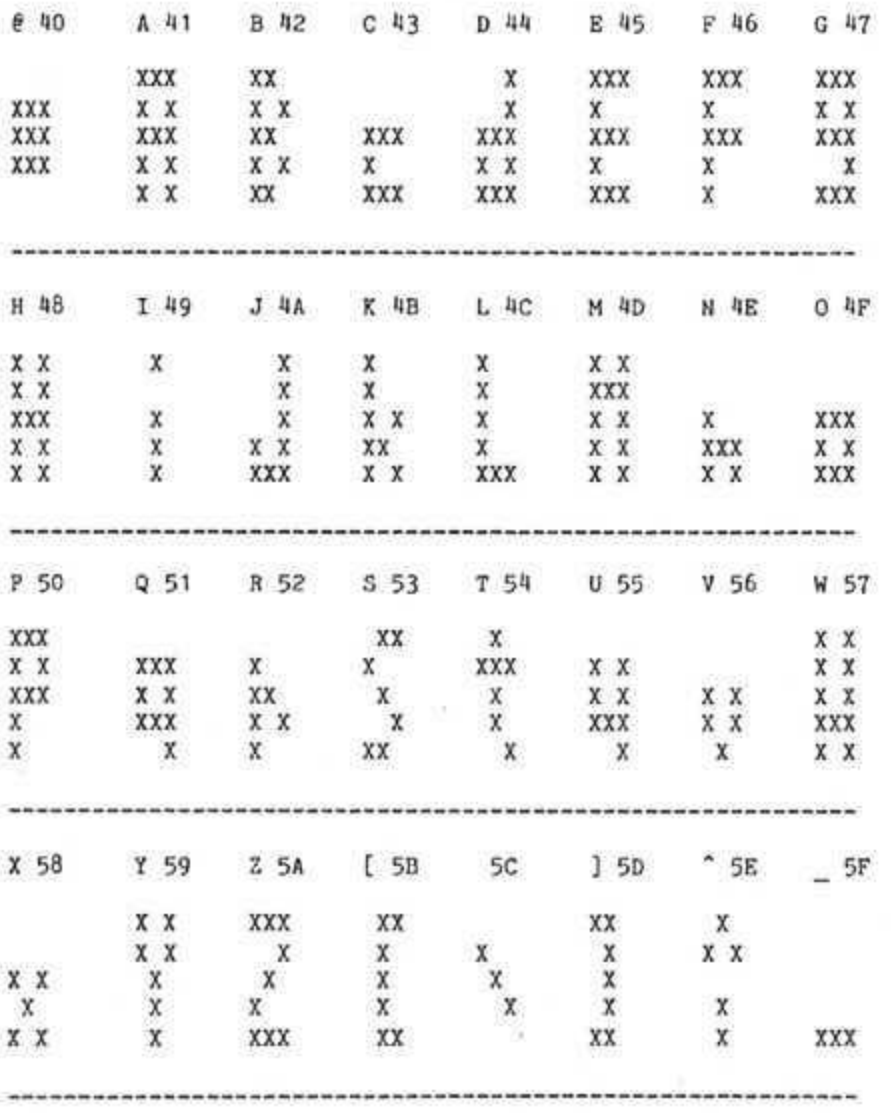
MagiCard Character Set

The MagiCard character display subroutine uses a subset of the "ASCII" character set to specify the character to be displayed. The missing ASCII codes 00 - 1F map to 40 - 5F and 60 - 7F map to 20 - 3F. The possible ASCII character codes and the corresponding character for each are given below:

ASCII Code	Character	ASCII Code	Character
20	blank	40	@
21	!	41	A
22	"	42	B
23	#	43	C
24	$	44	D
25	%	45	E
26	&	46	F
27	'	47	G
28	(	48	H
29	)	49	I
2A	*	4A	J
2B	+	4B	K
2C	,	4C	L
2D	-	4D	M
2E	.	4E	N
2F	/	4F	O
30	0	50	P
31	1	51	Q
32	2	52	R
33	3	53	S
34	4	54	T
35	5	55	U
36	6	56	V
37	7	57	W
38	8	58	X
39	9	59	Y
3A	:	5A	Z
3B	;	5B	[
3C	<	5C	\
3D	=	5D	]
3E	>	5E	^
3F	?	5F	_

On this and the next page, we illustrate how the MagiCard displays each of its characters on the TV screen.




@ 40
A 41
B 42
C 43
D 44
E 45
F 46
G 47
H 48
I 49
J 4A
K 4B
L 4C
M 4D
N 4E
O 4F
P 50
Q 51
R 52
S 53
T 54
U 55
V 56
W 57
X 58
Y 59
Z 5A
[5B
5C
] 5D
^ 5E
_ 5F

APPENDIX C

6502 INSTRUCTION SUMMARY

First Hex Digit	Second Hex Digit 0	1	2	3	4	5	6	7
0	BRK	ORA X)				ORA Z	ASL Z	
1	BPL	ORA)Y				ORA ZX	ASL ZX	
2	JSR	AND X)			BIT Z	AND Z	ROL Z	
3	BMI	AND)Y				AND ZX	ROL ZX	
4	RTI	EOR X)				EOR Z	LSR Z	
5	BVC	EOR)Y				EOR ZX	LSR ZX	
6	RTS	ADC X)				ADC Z	ROR Z	
7	BVS	ADC)Y				ADC ZX	ROR ZX	
8		STA X)			STY Z	STA Z	STX Z	
9	BCC	STA)Y			STY ZX	STA ZX	STX ZY	
A	LDY I	LDA X)	LDX I		LDY Z	LDA Z	LDX Z	
B	BCS	LDA)Y			LDY ZX	LDA ZX	LDX ZY	
C	CPY I	CMP X)			CPY Z	CMP Z	DEC Z	
D	BNE	CMP)Y				CMP ZX	DEC ZX	
E	CPX I	SBC X)			CPX Z	SBC Z	INC Z	
F	BEQ	SBC)Y				SBC ZX	INC ZX	

First Hex Digit	Second Hex Digit 8	9	A	B	C	D	E	F
0	PHP	ORA I	ASL Acc			ORA	ASL	
1	CLC	ORA Y				ORA X	ASL X	
2	PLP	AND I	ROL Acc		BIT	AND	ROL	
3	SEC	AND Y				AND X	ROL X	
4	PHA	EOR I	LSR Acc		JMP	EOR	LSR	
5	CLI	EOR Y				EOR X	LSR X	
6	PLA	ADC I	ROR Acc		JMP ()	ADC	ROR	
7	SEI	ADC Y				ADC X	ROR X	
8	DEY		TXA		STY	STA	STX	
9	TYA	STA Y	TXS			STA X		
A	TAY	LDA I	TAX		LDY	LDA	LDX	
B	CLV	LDA Y	TSX		LDY X	LDA X	LDX Y	
C	INY	CMP I	DEX		CPY	CMP	DEC	
D	CLD	CMP Y				CMP X	DEC X	
E	INX	SBC I	NOP		CPX	SBC	INC	
F	SED	SBC Y				SBC X	INC X	

Accumulator Instructions

operation	Addressing Mode							
	Imm	Abs	z-page	ind,x	ind,y	z,x	abs,x	abs,y
	I	A	Z	X)	)Y	ZX	X	Y
ADC	69	6D	65	61	71	75	7D	79
AND	29	2D	25	21	31	35	3D	39
CMP	C9	CD	C5	C1	D1	D5	DD	D9
EOR	49	4D	45	41	51	55	5D	59
LDA	A9	AD	A5	A1	B1	B5	BD	B9
ORA	09	0D	05	01	11	15	1D	19
SBC	E9	ED	E5	E1	F1	F5	FD	F9
STA		8D	85	81	91	95	9D	99

One-Operand Instructions

operation	Addressing Mode				
	Acc	Abs	z-page	z,x	abs,x
		A	Z	ZX	X
ASL	0A	0E	06	16	1E
DEC		CE	C6	D6	DE
INC		EE	E6	F6	FE
LSR	4A	4E	46	56	5E
ROL	2A	2E	26	36	3E
ROR	6A	6E	66	76	7E

Index Register Instructions

operation	Addressing Mode						
	Imm	Abs	z-page	z,x	Abs,x	Z,y	Abs,y
	I	A	Z	ZX	X	ZY	Y
CPX	E0	EC	E4				
CPY	C0	CC	C4				
LDX	A2	AE	A6			B6	BE
LDY	A0	AC	A4	B4	BC		
STX		8E	86			96	
STY		8C	84	94			

Miscellaneous Instructions

DEX CA	BCC 90	CLC 18	PHA 48	NOP	EA
DEY 88	BCS B0	SEC 38	PHP 08	BRK	00
INX E8	BEQ F0	CLD D8	PLA 68	BIT A	2C
INY C8	BMI 30	SED F8	PLP 28	BIT Z	24
TAX AA	BNE D0	CLI 58	RTI 40	JSR	20
TAY A8	BPL 10	SEI 78	RTS 60	JMP A	4C
TSX BA	BVC 50	CLV B8		JMP ()	6C
TXA 8A	BVS 70				
TXS 9A					
TYA 98					

APPENDIX D

"Life"--a sample program

When we looked for an idea for a typical moderately large sample program, the "Life" game invented by Prof. John Horton Conway of the University of Cambridge seemed a natural choice. Life was first presented by Martin Gardner in the Mathematical Games section of "Scientific American" in October 1970, then again in Febuary 1971 along with an introduction to cellular automata theory, from which the game was evolved. Since then the game has achieved a continuing popularity, with numerous articles appearing in such magazines as "Byte" and "Kilobaud", references in serious mathematical texts and sporadic newsletters.

Life consists of rules that govern the development of patterns of dots on a rectangular grid. Each dot is considered to be a "live cell"; a cell (or grid element) that is empty is considered to be "dead". Each cell has eight neighbors surrounding it. By counting the living neighbors of each cell (empty or not) and applying the following rules to all the cells in a grid at once, a new "generation" is formed.

1. Any live cell with 2 or 3 live neighbors survives.

2. Any live cell with 0 or 1 living neighbors dies (of loneliness).

3. Any live cell with 4 or more neighbors dies (of overcrowding).

4. Any empty (or dead) cell with exactly 3 live neighbors is "born" (comes to life).

Even patterns of only a few dots can evolve in surprising and interesting ways. Some patterns are static, 4 dots arranged adjacent to each other in a square is the simplest example. Some patterns oscillate, and return to their initial configuration after 2 to hundreds of generations in some cases. Three dots in a vertical line will alternate with 3 dots in a horizontal line, to make the simplest oscillator. A very simple pattern will often explode into chaotic activity before eventually settling down to static or oscillating components. Try 3 vertical dots (like the simple oscillater), with 2 dots added. Add a dot in contact with the center cell on the right, and another in contact with the bottom cell on the left. Position this pattern about 1/3 of the way from the left edge of the tv screen and below the center of the screen vertically; this will forestall the disruption that occurs when the pattern hits the edges of the (40

x 42 decimal) grid area.

Look especially for a small pattern of dots that wiggles along diagonally and "flys" to the edge of the screen. This is called a "glider" and is the simplest example of a traveling pattern. It is often interesting to follow the progress of parts of the debris of a developing pattern. For example, look for a pattern like a small 4 that appears as part of the above 5 dot pattern, which in isolation has an interesting life of its own.

To run Life, enter it into the computer and begin execution at F15E. Life is long, so disassemble it to check for key-in errors and save it on cassette if you have built the interface. While running, a slightly flickering "cursor" dot will appear in the center of the screen. It can be moved around by pressing keys on the left controller. L2 moves it up, L4 moves it left, L6 moves it right and L8 moves it down. L5 causes the dot at the cursor location to change from live to dead or vice versa. L0 causes the current pattern to be updated to the next generation. This will continue as long as L0 is held down. Notice that TV sync is lost during generation updating. If any left key (except L0) is held down, the cursor is suppressed and you can see what is "under" it. The cursor itself is not a live cell and its position has no affect on forming new generations. To clear the screen when it is cluttered, press "game reset" and run the program again.

We programmed Life just the way any user might. First we decided how the game should look and what compromises we would accept to make it easy to program the first version. Then we identified the major components and flow of the program, including sections that should be subroutines. This is often done with boxes and arrows, arranged in diagrams call flowcharts. Next we wrote out the program in text lines that would each be implementable as a few 6502 instructions. Names were used for storage locations. We examined the text for logical flaws and disagreements with our overall plan. Then we assigned addresses for the named storage locations. Next we wrote the right side of the listing, still using names for many storage locations, and label names for locations that were the targets of long or frequent jumps and branches. We then assigned F100 as the address of the first location of the program and counted bytes carefully through the code, assigning addresses to labeled instructions. Next we looked up the opcodes and wrote the left side of the listing (the actual Hex), filling in addresses at the same time. Finally, the long relative branches were calculated (using the Relc key) and written on the hex listing as it was being keyed in.

We disassembled (Ins Dump) to check for errors and saved the program on cassette. It ran the first time! Often even a short program will not work at first, but care in planning, especially with the early text version, will yield working programs much more often than the "hacker" approach favored by some programmers. If the program had been much longer, we would have tested it in parts. For example, the key input section can be run without the generation update code. In professional programming publications, techniques to produce correct programs quickly have received much attention recently under the name "structured programming".

Even if you enjoy this program, you will probably notice a number of things that you might want to improve, which we ignored in the

interest of a short and simple program. Examples include: maintaining sync (if not the picture) during generation updating, handling of the edges of the grid to really use those cells (perhaps with wrap around), counting the number of generations, plotting and moving whole patterns instead of single dots, and better controller capability (diagonal motion, joysticks). If you are really enthusiastic about Life, you may be interested in our "Lifecard" module which will soon be available. It features a 96 x 80 (decimal) game grid and display, all the improvements mentioned above and more.

Memory Usage: (also F300 - F3D1 is used as a copy of the display)

name	location	use
MZ	B0	Mask of bit for (H0,V-) position, also for cursor position.
AZ	B1	Address offset for (H0,V-) position, also for cursor position.
M+	B2	Mask of bit for (H+,V-) position, also temporary storage.
A+	B3	Address offset for (H+,V-) position.
CZ	B4	=0 if vertical middle cell at center is empty, else =FF.
C+	B5	=0 if vertical middle cell at right is empty, else =FF.
N-	B6	Number of neighbors in left 3 cells.
NZ	B7	Number of neighbors in center 3 cells, including target cell.
N+	B8	Number of neighbors in right 3 cells.
HC	B9	Horizontal cursor position (00 - 27).
VC	B9	Vertical cursor position (00 - 29).
DOT	BB	Value of dot at cursor position (on = FF, off = 00).

Note the "<" to mark places where the Relc key was used to calculate relative branches.

```
F100  A9 00    COPCL:  LDA I  00      ;Copy and clear subroutine
      85 8C            STA Z  8C      ;set address
      A9 F7            LDA I  F7
      85 8D            STA Z  8D      ;for STPM
      A0 D2            LDY I  D2
F10A  B9 FFEF  L1:      LDA Y  EFFF
      88                 DEY          ;Copy display to
      20 F7FF            JSR    STPM: ;4th page of memory
F111  D0 F7              BNE    L1:
      A9 F4            LDA I  F4      ;Reset STPM address
      85 8D            STA Z  8D      ;to display.
      A9 00            LDA I  00
      A0 D2            LDY I  D2
F11B  88       L2:       DEY
      20 F7FF            JSR    STPM: ;Clear display.
      D0 FA              BNE    L2:
      60                 RTS
F122  A5 B2    NXH:    LDA Z  M+:     ;Set up for next point.
      85 B0            STA Z  MZ:     ;MZ = M+
```

```
      A5 B3           LDA Z  A+:
      85 B1           STA Z  AZ:      ;AZ = A+
      A5 B7           LDA Z  NZ:
      85 B6           STA Z  N-:      ;N- = NZ
      A5 B8           LDA Z  N+:
F130  85 B7           STA Z  NZ:      ;NZ = N+
      A5 B5           LDA Z  C+:
      85 B4           STA Z  CZ:      ;CZ = C+
      20 BEFF         JSR    CALP:    ;Get V-,H+ position,
      85 B2           STA Z  M+:      ;we will use "secret
      84 B3           STY Z  A+:      ;knowledge": successive bytes
      A2 00           LDX I  00       ;plot vertically down display.
      86 B5           STX Z  C+:      ;C+ = 0
F141  39 00F3         AND Y  F300     ;Test H+,V- position
      F0 01           BEQ    01       ;and count it.
      E8              INX
      A5 B2           LDA Z  M+:
      39 01F3         AND Y  F301     ;Test H+,VZ position
      F0 03           BEQ    03       ;and count it
      C6 B5           DEC Z  C+:      ;for both C+ (-1)
F150  E8              INX              ;and N+.
      A5 B2           LDA Z  M+:
      39 02F3         AND Y  F302     ;Test H+,V+ position
      F0 01           BEQ    01       ;and count it.
      E8              INX
      86 B8           STX Z  N+:      ;N+ =   of right neighbors.
      E6 8E           INC Z  8E       ;H+ = H+ + 1
      60              RTS
F15E  20 00F1 ENTRY:  JSR    COPCL:   ;Clear screen.
      A9 14           LDA I  14
      85 B9           STA Z  HC:      ;Set horiz and vertical
      85 BA           STA Z  VC:      ;cursor position.
F167  A5 B9   KEEP:   LDA Z  HC:
      85 8E           STA Z  8E
      A5 BA           LDA Z  VC:
      85 8F           STA Z  8F
      20 BEFF         JSR    CALP:    ;Calculate cursor position
F172  85 B0           STA Z  MZ:      ;and save.
      84 B1           STY Z  AZ:
      A2 00           LDX I  00
      39 00F0         AND Y  F000     ;Test cursor position.
      F0 01           BEQ    01       ;00 => dot off.
      CA              DEX             ;FF => dot on.
      86 BB           STX Z  DOT:     ;Save cursor block status.
F180  20 6FFA KREAD:  JSR    DSPL:    ;Display and read keys.
      A4 B1           LDY Z  AZ:
      A5 B0           LDA Z  MZ:
      59 00F0         EOR Y  F000     ;Complement dot at
      20 F7FF         JSR    STPM:    ;cursor position.
      A5 8A           LDA Z  8A       ;If no left key down,
      30 EF           BMI    KREAD:   ;loop on display.
F191  A5 B0           LDA Z  MZ:
      49 FF           EOR I  FF
      39 00F0         AND Y  F000
      85 B2           STA Z  M+:      ;Save with bit masked off.
      A5 B0           LDA Z  MZ:
      25 BB           AND Z  DOT:     ;Re-establish original
      05 B2           ORA Z  M+:      ;dot value at cursor
F1A0  20 F7FF         JSR    STPM:    ;position.
```

```
      A5 82              LDA Z  82
      C9 05              CMP I  05
      D0 09              BNE    09
      A9 FF              LDA I  FF       ;Complement final
      45 BB              EOR Z  DOT:     ;value of dot at
      85 BB              STA Z  DOT:     ;cursor position.
      4C 80F1            JMP    KREAD:
F1B2  C9 02              CMP I  02
      D0 08              BNE    08
      A5 BA              LDA Z  VC:
      F0 AD  <           BEQ    KEEP:
      C6 BA              DEC Z  VC:      ;Move cursor up.
      10 A9  <           BPL    KEEP:    ;Unconditional branch.
      C9 04              CMP I  04
F1C0  D0 08              BNE    08
      A5 B9              LDA Z  HC:
      F0 A1  <           BEQ    KEEP:
      C6 B9              DEC Z  HC:      ;Move cursor left
      10 9D  <           BPL    KEEP:    ;Unconditional branch.
      C9 06              CMP I  06
      D0 0A              BNE    0A
      A5 B9              LDA Z  HC:
F1D0  C9 27              CMP I  27
      10 93  <           BPL    KEEP:
F1D4  E6 B9              INC Z  HC:      ;Move cursor right.
      10 8F  <           BPL    KEEP:    ;Unconditional branch.
      C9 08              CMP I  08
      D0 08              BNE    08
      A5 BA              LDA Z  VC:
      C9 29              CMP I  29
F1E0  10 85  <           BPL    KEEP:
      E6 BA              INC Z  VC:      ;Move cursor down.
      A5 8A              LDA Z  8A       ;If left zero not down,
      D0 41  <           BNE    LINK:    ;bypass generation update.
      20 00F1            JSR    COPCL:   ;Save display and clear.
      85 8F              STA Z  8F       ;Zero V-.
F1ED  A9 00    NEXTV:    LDA I  00
      85 8E              STA Z  8E       ;Zero H+.
F1F1  20 22F1            JSR    NXH:     ;Initialize for sweep
      20 22F1            JSR    NXH:     ;of a line of dots.
F1F7  20 22F1  NEXTH:    JSR    NXH:     ;Update for next dot.
      8A                 TXA             ;Get N+.
      18                 CLC
      65 B7              ADC Z  NZ:      ;Calculate number
      65 B6              ADC Z  N-:      ;of neighbors.
F200  65 B4              ADC Z  CZ:      ;N+ + N- + NZ - CZ
      C9 02              CMP I  02
      30 15              BMI    NEXT:    ;Skip if   < 2.
      C9 03              CMP I  03
      F0 06              BEQ    06       ;Plot if   = 3.
      10 0F              BPL    NEXT:    ;Skip if   > 3.
      A5 B4              LDA Z  CZ:      ;Must be   = 2,
      F0 0B              BEQ    NEXT:    ;skip if not occupied.
F210  A5 B0              LDA Z  MZ:      ;Plot a
      A4 B1              LDY Z  AZ:      ;dot at
      C8                 INY             ;the current position.
      19 00F0            ORA Y  F000
      20 F7FF            JSR    STPM:
F21B  A5 8E  < NEXT:     LDA Z  8E       ;Loop if H+ < 28.
```

```
      C9 28              CMP I  28
      30 D6  <           BMI    NEXTH:
F221  E6 8F              INC Z  8F       ;V- = V- + 1.
      A5 8F              LDA Z  8F       ;Loop if V- < 28.
      C9 28              CMP I  28
      30 C4  <           BMI    NEXTV:
F229  4C 67F1  LINK:     JMP    KEEP:    ;Generation complete.
F22C
```

APPENDIX E

Constructing and Connecting a Cassette Interface

Sections 6 and 7 of Chapter 4 describe the operations needed to write and read programs on cassette tape. In this appendix we give details on construction and connection for the cassette interface. A schematic of the interface and a list of parts will be found at the end of this appendix.

Theory of Operation

The interface circuit is simply a signal buffer. The diode in the monitor input section clips the negative going parts of the audio signal from the cassette player. The resistors provide impedance matching and isolation. The I.C. sections are Schmitt triggers. A Schmitt trigger takes a slow rough rising or falling signal, typical of the cassette monitor output, and makes a clean abrupt logic transition. These transitions are read by the 6502 processor from the 6532 (RIOT) I/O port A.

The pulses are recorded with three different intervals. The shortest interval represents a "1" bit, the medium interval represents a "0" and any longer interval is a "fill" or separator signal. Each byte is recorded as 8 bits, least significant first, plus a 9th parity bit. Each group of 9 bits is seperated by a "fill" signal. The parity bit is set "on" if the preceeding 8 bits have an odd number of "1" bits. This way of calculating the value of the parity bit is called "even parity", because in each 9 bit group an even number of bits will always be "on". When the tape is read, the parity of each group is checked and the reading stops if any group fails to have even parity.

The part of the circuit connected to the cassette recorder microphone input simply isolates the cassette recorder from the game. The resistors provide a properly attenuated and impedence matched signal for the recorder. The jumper which connects points (b) and (c) in the schematic is in the proper position for most cassette recorders. Most inexpensive cassette recorders play back a signal that is inverted from the one that was recorded. If your recorder's output is not inverted, wire the jumper between points (a) and (b) to provide the extra inversion needed.

What Kind of Cassette Recorder to Use

Two inexpensive cassette recorders that work well with the interface are the General Electric model 3-5001A and the Panasonic model RQ-2765. The Panasonic costs about $40 at discount and features a tape position counter, which makes it easier to find a program if

several have been recorded on one side of a cassette tape. The GE at about half the price is bulkier and lacks the tape counter, but works satisfactorily. Both of these recorders work with the jumper as shown in the schematic. Many portable cassette recorders have a battery eliminator, but these vary in design and may damage your game. Always use a battery powered cassette recorder, and operate it only on its batteries.

Construction

With only one I.C. and a few passive components, this circuit can be constructed by a variety of techniques. A small piece of circuit board with holes on 1/10 inch centers is a convenient base for the components. If you use a socket for the I.C., construction will be easier and the I.C. is much less likely to be damaged. The completed board can be placed in a small plastic box, with the switch mounted on a hole in the box. Be sure to provide some strain relief for the cables so that they won't be pulled loose by handling.

The connection to the game itself can be made in one of two ways. One way is to use male and female controller connectors to connect the left controller to the interface and then through to the left controller socket on the game console. The other way is to simply cut into the cable on one of your keyboard controllers and connect directly to the wires there. You will probably find it most convenient to cut into the cable fairly near the connector to the game console. The box containing the interface can be connected directly to the controller cable. A short connecting cable, perhaps with a multi-prong connector to allow it to be disconnected, will make a more convenient package.

On the plugs connecting to the cassette recorder, the shaft is ground and the tip carries the signal. On the 1N914 diode a band will be found near one end of its small glass body. The end opposite the band is to be connected to the ground part of the circuit. The 15 Ohm resistor should be rated at a minimum of one Watt to handle the largest possible signals from the recorder. The switch is a DPST (double pole single throw) though a DPDT can be used, leaving the extra terminals on the switch unconnected. The switch is shown open (or off) in the schematic. This is the switch position that disconnects the interface when it is not actually in use.

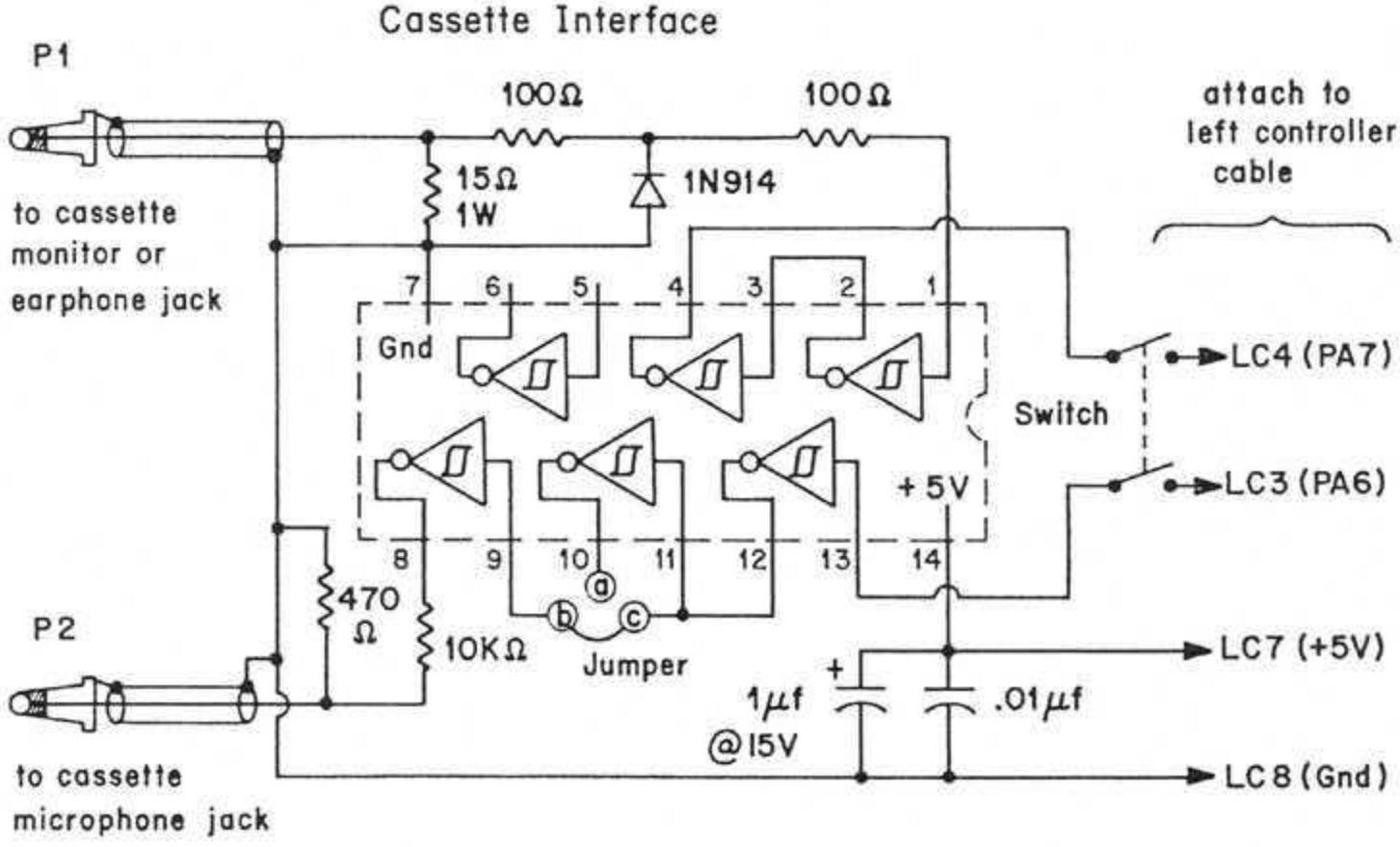
Connecting to the Video Computer System

The cassette interface is very easy to connect. Be sure to connect it with the game and cassette recorder both turned off. If it is attached directly to one of the controller cables, make sure this is the left controller and is plugged into the left controller socket on the game console. If the interface has its own connectors, connect it to the left controller socket and be sure that the controller you plug into the interface is on your left. Next plug in the two cables to the cassette recorder. Since the two plugs are identical, you will have to label the cables to avoid confusion. Before turning on the game, be sure the cassette interface switch is off. Most problems with the keyboard controllers and cassette interface are due to misconnected cables or to the cassette interface switch being inadvertently left on.

Parts List

I.C.	1- 74LS14	Six inverting Schmitt triggers, in a 14 pin DIP (Widely available by mail order.)
Resistors	2- 100 Ohm	RS 271-1311
	1- 470 Ohm	RS 271-1317
	1- 10k Ohm	RS 271-1335
	1- 15 Ohm 1 Watt	Can use 2 27 Ohm 1/2 watt in parallel RS 271-006
Capacitors	1- .01 microfarad	RS 272-131
	1- 1.0 microfarad	RS 272-1419 Observe polarity of this capacitor.
Diode	1- 1N914	switching diode RS 276-1122
Switch	1- submini DPDT	RS 275-614 A variety of switches will work.
P1 and P2	2- plugs	1/8 inch miniature plugs RS 274-287
Misc.		Plastic box, shielded cable (to cassette recorder), 14 pin DIP I.C. socket (RS 276-1999 or RS 276-1993), solder, wire, etc.
Optional		Male and Female 9-pin controller plugs: DE9P and DE9S, available from: California Digital Box 3097B Torrance, California 90503

Note: Part numbers labeled with "RS" are distributed by Radio Shack.



IC is shown viewed from the top

Owner Information

Your MagiCard has been thoroughly tested at the factory and should provide years of programming enjoyment. It is guaranteed for 90 days against defects in materials or workmanship. If you suspect your MagiCard is defective, however, you should write to us first describing the symptoms--most likely you are not using it properly and a few words of advice will get you up and running. We of course cannot be responsible for any damage due to improper use of the MagiCard.

Repairs after the warranty period has expired will be made at cost. Again, don't send us your MagiCard before notifying us of the nature of the problem and getting our authorization to return it for repair.

We will be happy to answer any questions about MagiCard operation or programming problems. Please include a self-addressed stamped envelope to facilitate a reply. And let us know about any particularly interesting programs you write and would like to share with other MagiCard users. We intend to distribute sample programs for a nominal fee.

Finally, note that Atari and Video Computer System are trademarks of Atari, Inc. The MagiCard will also work in Sears TeleGame, a trademark of Sears Roebuck, Inc. A patent is pending on the MagiCard design.